8-1-1993

Channel routing: Efficient solutions using neural networks

Taj-ul Islam

CHANNEL ROUTING:

EFFICIENT SOLUTIONS

USING
NEURAL NETWORKS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

OF THE ROCHESTER INSTITUTE OF TECHNOLOGY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER ENGINEERING

By

Taj-ul Islam

August 1993

Title of Thesis:

Channel Routing:

Efficient Solutions Using Neural Networks

I, Taj-ul Islam, hereby **grant permission** to the Wallace Memorial Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: August 16, 1993.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science in Computer Engineering.

Professor Tony Chang, Ph.D.
Department of Computer Engineering
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science in Computer Engineering.

Professor Peter G. Anderson, Ph.D.
Department of Computer Science

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science in Computer Engineering.

Professor Roy Czernikowski, Ph.D.
Department of Computer Engineering

Abstract

Neural network architectures are effectively applied to solve the channel routing problem. Algorithms for both two-layer and multilayer channel-width minimization, and constrained via minimization are proposed and implemented. Experimental results show that the proposed channel-width minimization algorithms are much superior in *all* respects compared to existing algorithms. The optimal two-layer solutions to most of the benchmark problems, not previously obtained, are obtained for the first time, including an optimal solution to the famous Deutch's *difficult* problem. The optimal solution in four-layers for one of the benchmark problems, not previously obtained, is obtained for the first time. Both convergence rate and the speed with which the simulations are executed are outstanding. A neural network solution to the constrained via minimization problem is also presented. In addition, a fast and simple linear-time algorithm is presented, possibly for the first time, for coloring of vertices of an interval graph, provided the line intervals are given.

Acknowledgements

I wish to thank Professor Tony Chang for his constant guidance in writing this thesis, and, Professors R. Czernikowski and P. Anderson for thoroughly reading this thesis. I am profoundly grateful to my wife Chin for her unflagging devotion and support. And, to my daughter Roxanna, I wish to thank her for her cheerful countenance and extraordinary patience in tolerating my constant absence.

Contents

List of Tables

List of Figures

Chapter 1

Channel-Width Minimization

1.1 Introduction

Channel routing is a key problem in the automatic layout of electronic circuits, and was first introduced in 1971 by Hashimoto and Stevens [1], as part of their layout method for printed circuit designs. It is, however, equally applicable to VLSI circuit design. The *channel* is a rectangular region with terminals on two sides, as shown in Fig. 1.1. An imaginary grid is superimposed on the channel. The horizontal lines are called *tracks*, whereas, the vertical lines are called *columns*. All the terminals bearing the same label constitute a *net*, and all the terminals of a *net* must be connected together by conductors, which are constrained to lie on the grid. Terminals with the '0' label are unconnected. A net may contain any number of terminals. Nets with two terminals are called *two-point* nets, whereas, nets with more than two terminals are called *multipoint* nets. Some models allow limited permutation of the terminals, but generally it is assumed that the net terminals are fixed, which is the model we will assume. A valid channel routing is obtained by making sure that there are no shorts

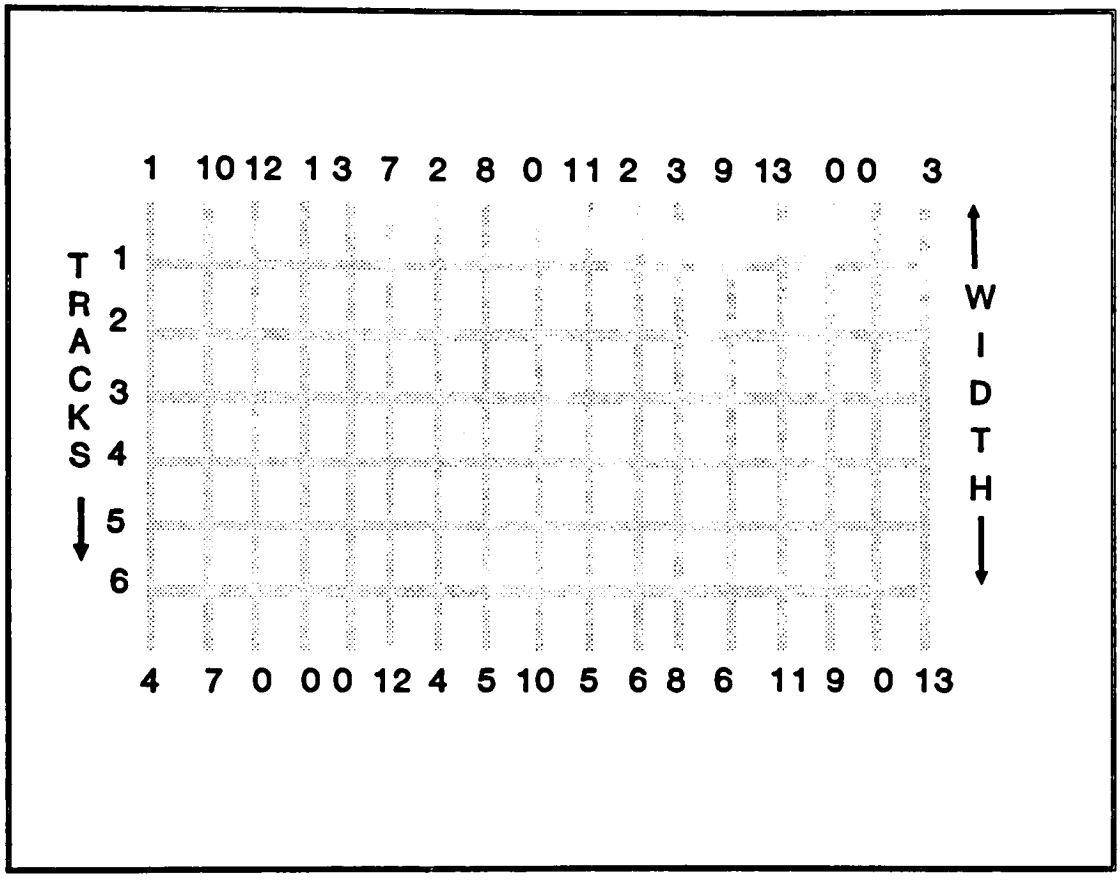


Figure 1.1: A channel with terminals on both sides are shown here. The interconnection region has 17 columns and six tracks available for routing. Terminals with the same non-zero number have to be electrically connected. The terminal value '0' is used for no connection. In rectilinear routing, an imaginary grid is assumed on which the conductors connecting the nets are deposited.

between two different nets. The goal of channel-width minimization is to produce a legal routing with the *smallest* number of tracks. As an example of a problem instance, we will consider the net specification shown in Fig. 1.2.

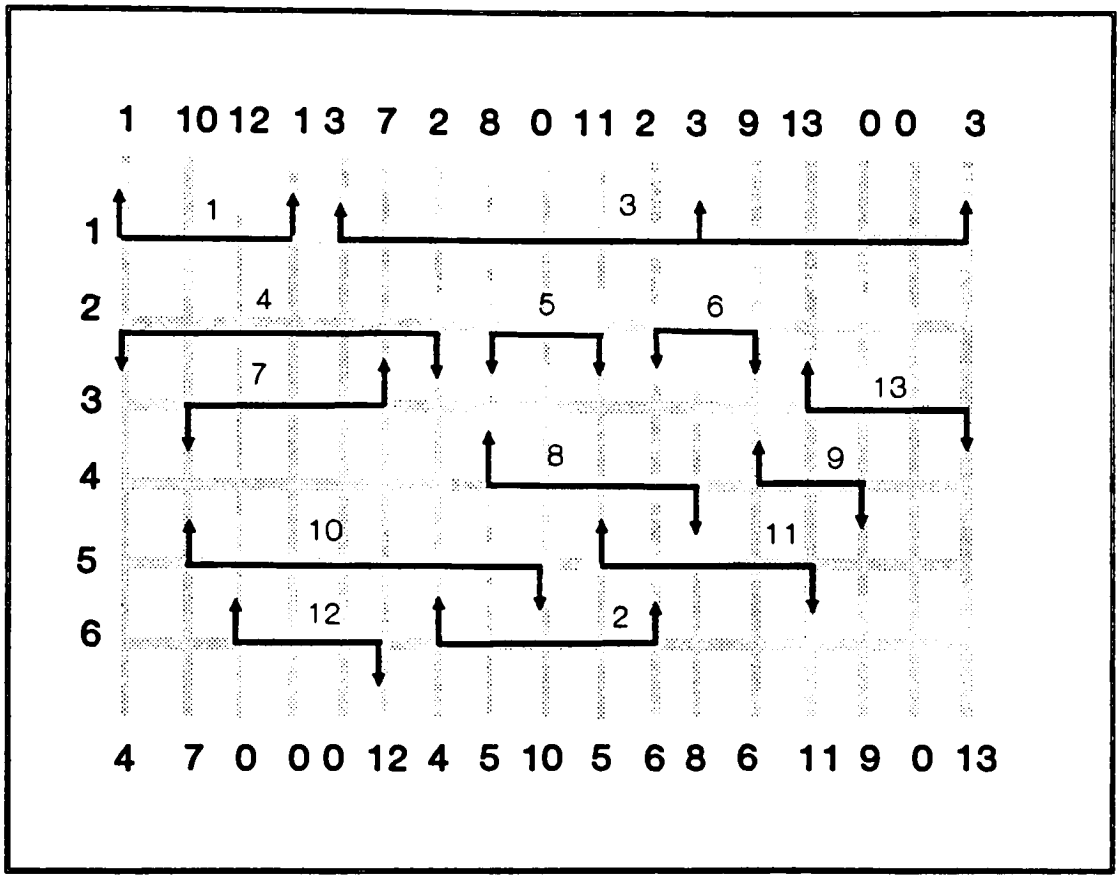


Figure 1.2: There are thirteen nets in this example. A valid routing will not short two different nets. Two layers are available for routing. In the Manhattan model, each layer is assigned to carry conductors in one direction only. Therefore, when a conductor changes direction, it has to switch layer, which is done by a *via* or a *cut*. For each net n, each of its terminals should be connected to the terminal strip indicated by the arrow with the terminal labelled n.

1.2 Wiring Models

Most channel routing models require two or more wiring layers. In this section, a two-layer model is assumed. Each layer is a two- dimensional plane on which conductors can be deposited along grid lines. For example, for two-layer VLSI channel designs, the conductor on one layer could be metal, and the conductor on the other layer could be polysilicon. Obviously, the conductors are separated by some non-conducting material like silicon dioxide. When a wire changes layer, it does so only at a grid point. This connection between two layers is called a *via* or a *contact.* The two most popular rectilinear wiring models are the *Manhattan* model and the *knock-knee* model. Another model, which is not commonly applicable in practice is the *river routing* model.

River Routing Model In this model different nets are required to be wired as vertex disjoint paths. In other words, it allows planar, one-layer routing only.

Manhattan Model This is a well-studied two-layer model, in which, each layer is assigned to carry conductors in a single direction. Consequently, it is also called a *directional* model. Segments on different layers can cross perpendicularly, and overlap along their lengths is obviously not allowed by the model, since each layer can carry conductors in a single fixed direction only. Whenever, a path changes direction, it must switch layers using a *via.* The Manhattan model solution of the problem presented in Fig. 1.2, is shown in Fig. 1.3.

Knock-Knee Model In this model, each layer can carry conductors in both horizontal and vertical directions. However, (conducting) segments on two separate layers belonging to two distinct nets cannot overlap along their lengths. The model also requires that whenever a corner is shared between two paths, they

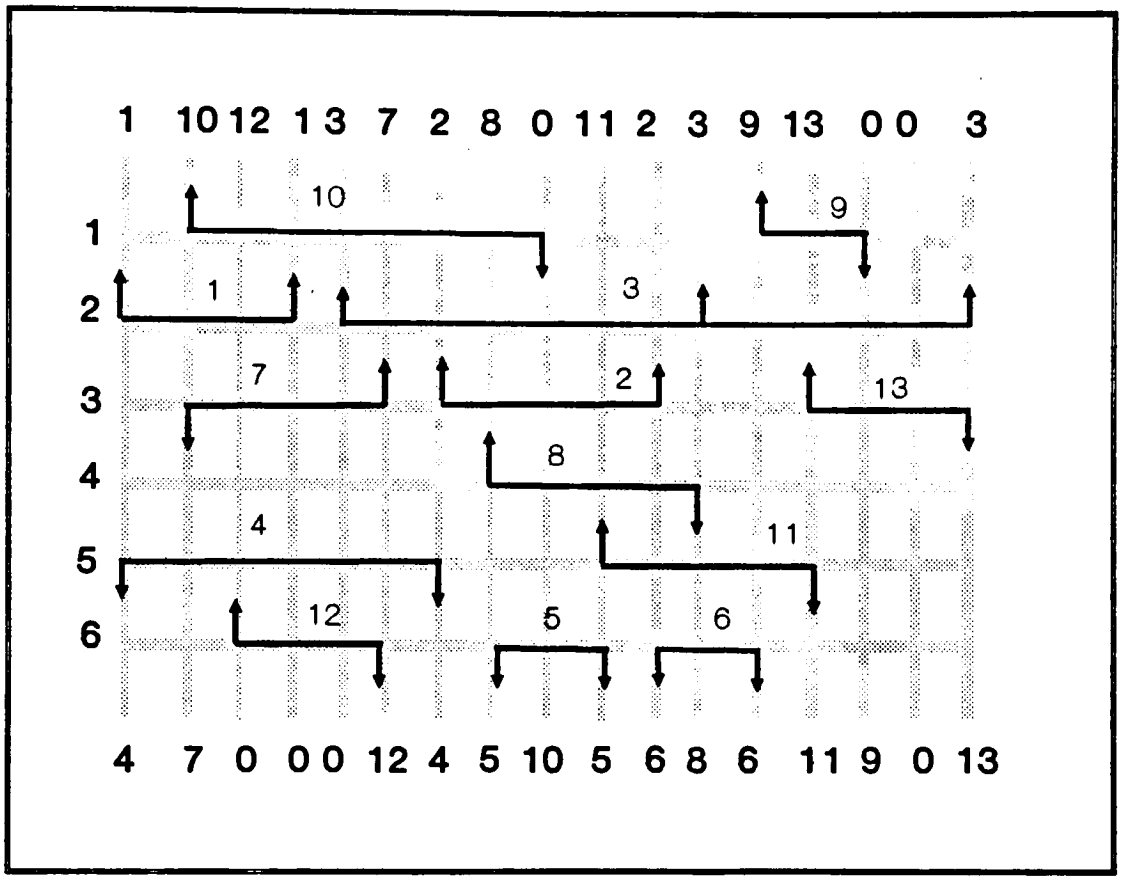


Figure 1.3: Shown above is the solution, in the *Manhattan* model, of the problem instance shown in Fig. 1.2. A legal solution must not have short-circuits. No vertical segment should cross another vertical segment. Similarly, no horizontal segment should cross another horizontal segment. Vias are not drawn above but exist at each corner, where the path changes direction. There are 27 vias.

be on different layers. In this model, the path of each wire and the layer of each wire must be determined. It is not a routing model that uses a specific number of layers. The model requires that edge-disjoint paths be used for distinct nets. Often the problem is broken up into two parts: (*a*) a set of edge-disjoint paths is found that interconnects the given nets and uses a given number of tracks (or minimizes the number of tracks) — the knock-knee wire layout; (*b*) given a knock-knee wire layout, a legal assignment is found using a given number of layers or minimizing the number of layers.

In the *restricted* channel routing model, the horizontal segment of each net is restricted to a single track. In other words, the horizontal segment of a net may not be split to reside on two different tracks. The *restricted* model is also called a *jog free* model. A *jog* is a vertical segment of wire that connects two horizontal segments of the same net assigned to two different tracks. A *dogleg* is a *jog* that exists in a column which contains a terminal for that net. Generally, the *unrestricted* models allow a net to be split only at a column that contains a terminal for that net, although optimal solutions are sometimes obtained only if jogs are allowed at any arbitrary column. Introducing *jogs* or *doglegs* can reduce the number of tracks needed by a routing, and, in some cases, can make the routing possible, as shown in Fig. 1.4.

Any *jog free* channel-width minimization algorithm can easily produce a routing with doglegs. This is done simply by first preprocessing the data, such that each *multipoint* net is converted to two or more two-point nets, as shown in Fig. 1.5. Consequently, doglegging is not considered in this thesis.

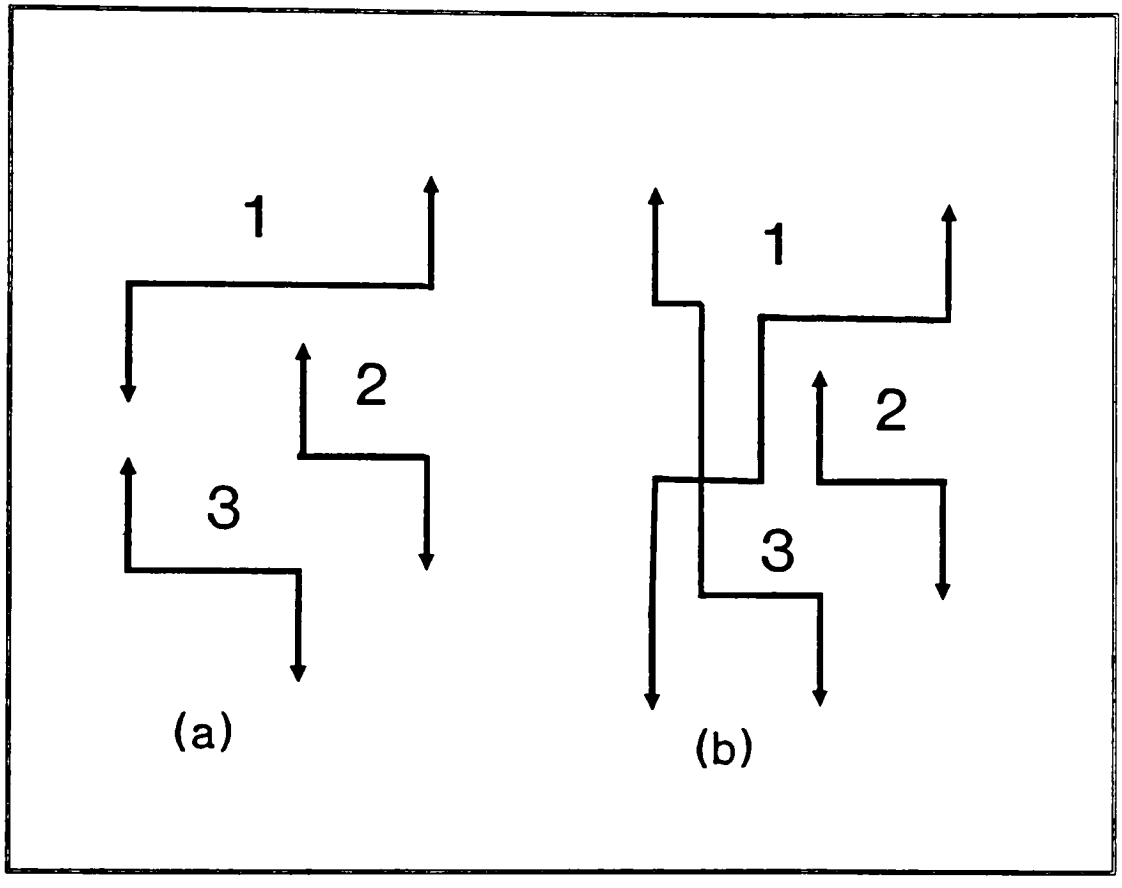


Figure 1.4: The routing specification given in (a) has a vertical conflict cycle. If we use the *restricted* model and thereby constrain each horizontal segment to reside on a single track, then the routing problem cannot be solved. In (b), both nets 1 and 3 have been *jogged* to make the routing possible.

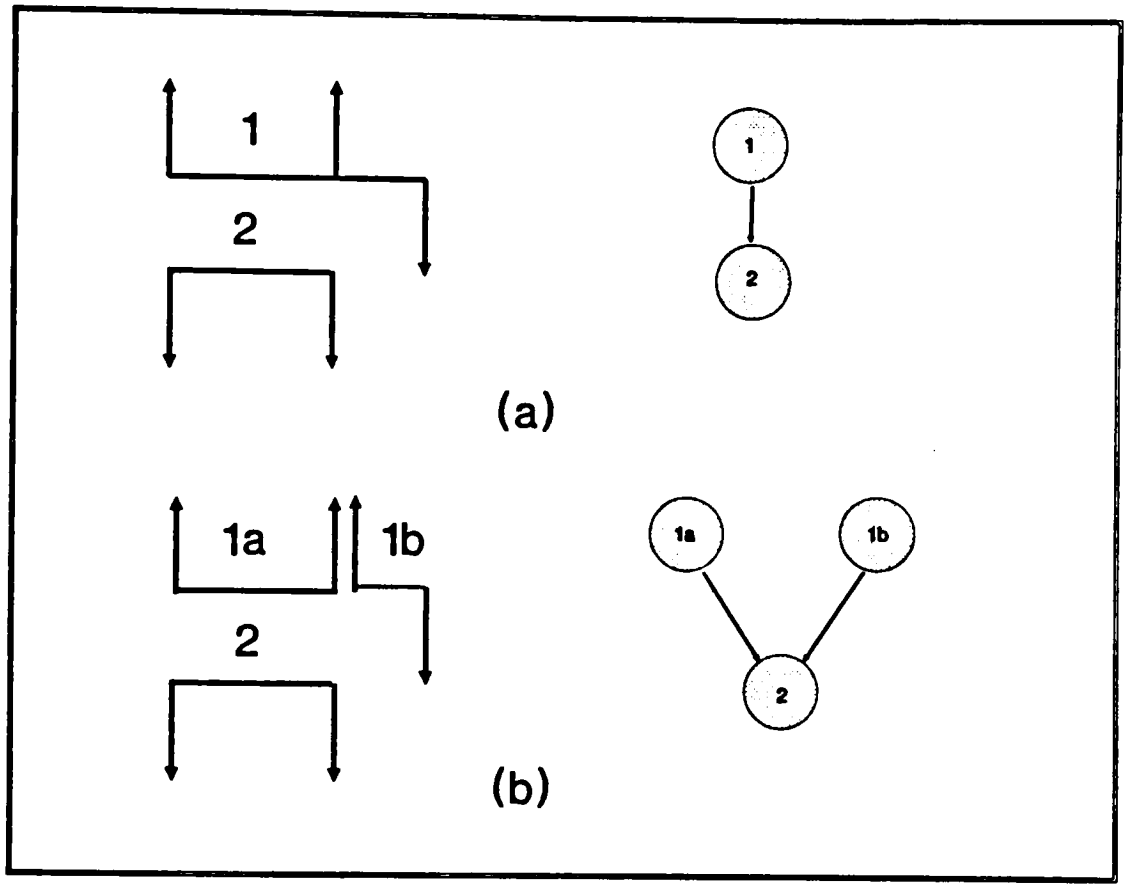


Figure 1.5: Any channel router can incorporate doglegging by simply preprocessing the data. In (a), the original problem is shown along with its vertical constraint graph, which is just a precedence graph indicating that net 1 must precede net 2. Each k-point ($k > 2$) net, is split at each of the internal terminals giving $k-1$ 2-point nets, as shown in (b). The vertical constraint graph is accordingly updated.

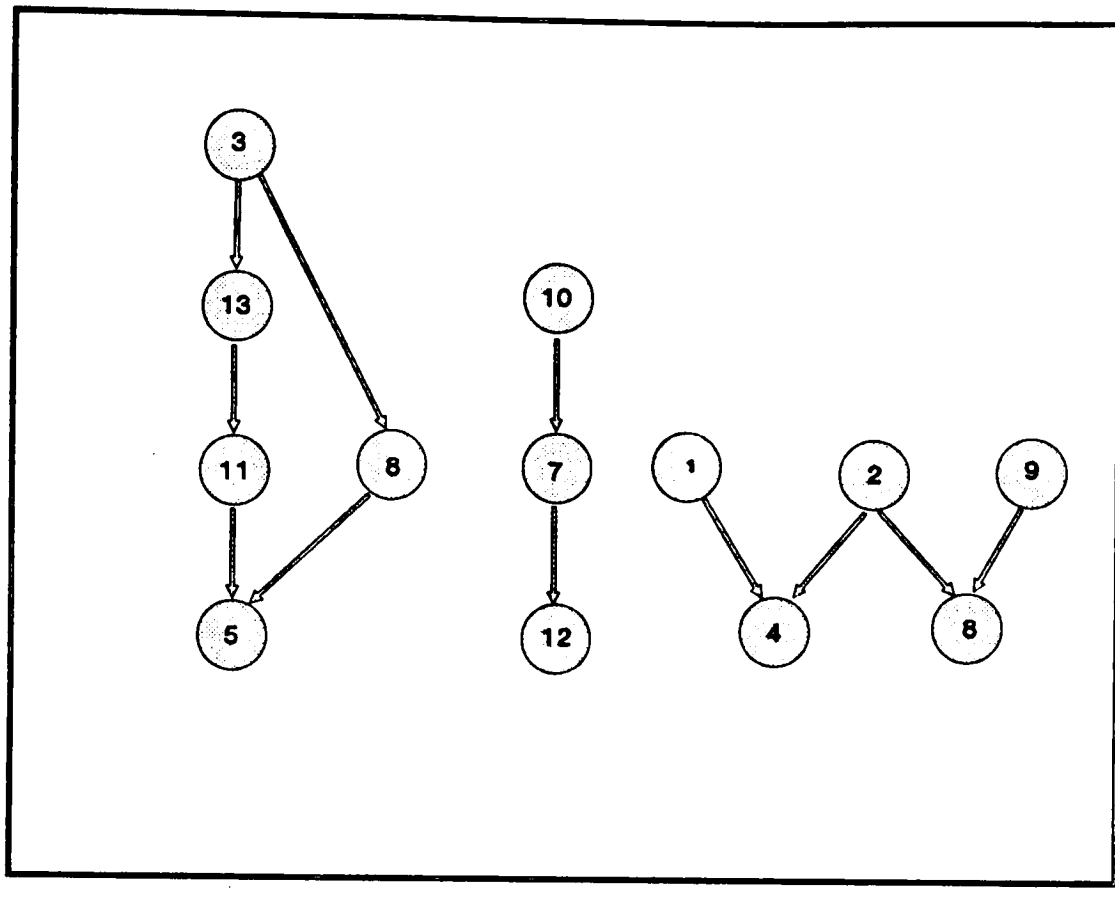


Figure 1.6: The vertical constraint graph for the problem instance shown in Fig. 1.2. The length of the longest path in the vertical constraint graph is four.

1.3 The Vertical Constraint Graph

Consider the problem instance shown in Fig. 1.2 again. It is obvious that net *13* must precede net *11*. Such considerations lead to precedence relations among the nets. These relations are compactly expressed by the *vertical constraint* graph. For the problem instance of Fig. 1.2, the vertical constraint graph is shown in Fig. 1.6. The length of the longest chain in the vertical constraint graph, which in this case is four, is an indication that at least four tracks are required. It is the vertical constraint graph which makes the channel routing problem difficult. If there are cycles in the

vertical constraint graph, then it must be removed by jogging as mentioned earlier. So, generally, the vertical constraint graph is regarded as *cycle-free*, which consequently makes it a directed *acyclic* graph. As is the case with all precedence graphs, the vertical constraint graph imposes a *partial order* on the set of nodes (nets).

1.4 The Horizontal Constraint Graph

The *horizontal constraint* graph shows what nets cannot be placed on the same track. Each node in the graph corresponds to a net, and two nodes have an edge between them if there is a horizontal overlap (even if it is just the extremities) between the corresponding nets, which obviously precludes them from occupying the same track. This graph is the well-known *interval graph.* The *coloring* of the nodes of this graph corresponds to the track assignment of the nets, provided the vertical constraint graph is ignored. Known methods [2] for coloring an interval graph are efficient with time complexity $\mathcal{O}(n \lg n)$, but complicated. A **new algorithm** is presented here which is much more efficient and easily implemented. The reason for this efficiency and simplicity is that the new algorithm processes the description of the line intervals directly, rather than the interval graph. It can color and find all the *dominant* cliques (maximal complete subgraphs) of the interval graph. If only two-point nets are involved, then it requires only one pass over the channel, column by column; otherwise, it requires two passes. In the latter case, the first pass determines the beginning and ending column of each net and the second pass determines all the dominant cliques and the coloring of the interval graph. It is also possible to make a pass over the order of occurrence of the nets and make the process even more efficient because generally there are there are fewer nets than columns. But this requires an intermediate array, which we have named *NetOrder* and is shown in Fig. 1.7 for the problem instance

```
 1  4 10   7 12  -1   3  -7 -12   2 -4  8  5  -10 11
-5  6 -2  -8  9   6  -6  13 -11  -9 -3 -13
```

Figure 1.7: The *NetOrder* array for the problem shown in Fig. 1.1. A positive integer n indicates the beginning of the net n and a negative integer $-n$ indicates the end of the net n. To construct this array, the columns of the channel are scanned from left to right. If the beginning of a net n is encountered, n is stored in the array. If the end of a net is encounterd $-n$ is stored in the array. If on the same column one net s starts and another net e ends, then s is stored first in the array and then $-e$. For multipoint nets, the starting column and ending column of each net needs to be known in advance. Vertical nets (zero extent) should be excluded from being included in the *NetOrder* array.

---

given in Fig. 1.1.

The algorithm described in Fig. 1.8, obviously makes one pass over the nets, although there are nested loops. The start and end of each net (excluding vertical nets, which should not be in *NetOrder*), is examined exactly once. Deleting an element $x \in S$ can also be done in constant time provided the position of x within S is known. The element x is deleted by replacing it with an element y, if any, located at the extreme end of set S. The position of y is then updated. All this can be done in constant time.

The *clique number* is the size (number of vertices) of the largest clique and, for an interval graph, it is also the *chromatic number*, i.e., the number of colors needed to color the vertices of the graph, without adjacent vertices ending up with the same color. The *clique number* is also called the *density* of the channel and is simply the maximum number of overlapping nets at some column. The *maximal cliques* for the problem instance shown in Fig 1.1 is shown in Fig. 1.9, obtained with the algorithm given in Fig. 1.8. A simple lower theoretical bound on the number of tracks needed

```
/* Find dominant cliques and vertex coloring (track
assignment of nets) of an interval graph. It is assumed that the
array NetOrder has been found already in previous pass
over the channel. */

    MaxColor = 0; /* Maximum number of colors used */
    Top = 0; /* Empty Available Color Stack*/
    CliqueSize = 0;
    for(i = 1; i <= NIntervals; ){
       for ( ;NetOrder[i] >0; i++){
          n = NetOrder[i];
          Clique[++CliqueSize] = n;
          PosInSet[n] = CliqueSize;
          if (Top == 0)
              color[n] = MaxColor++;
          else
              color[n] = AvailColorStack [Top--];
       }
       PrintArray(Clique, CliqueSize);

       for ( ;i <= NIntervals && NetOrder[i] < 0; i++){
          n = -NetOrder[i];
          AvailColorStack[++Top] = n;
          DeleteElemFromSet(Clique, &CliqueSize, PosInSet, n);
       }
    }
```

Figure 1.8: A fast linear-time $\Theta(N)$ algorithm for track assignment of nets, where N is the number of nets. It makes one single pass over the *NetOrder* array. When a new net is encountered it is placed in the current clique and a color is obtained from the available color stack if the stack is not empty, otherwise, a new color is generated. Upon encountering the first end of a net k, the clique is maximal. Net k and consecutively ending nets are deleted from the clique and their colors are made available again. This process is repeated until there are no more nets. The *DeleteElemFromSet* procedure is easily done in constant time. Note that the $\Theta(N)$ time complexity does not include the time for *PrintArray*.

Clique	Nets in Clique
1	1 4 10 7 12
2	3 4 10 7 12
3	3 4 10 2
4	3 5 10 2 8
5	3 5 11 2 8
6	3 6 11 2 8
7	3 6 11 9
8	3 13 11 9

Figure 1.9: Maximal (dominant) cliques obtained with the algorithm described in Fig. 1.8 for the problem instance given in Fig. 1.1. Each *maximal* clique is also called a *zone*.

---

is the larger of the channel density and the length of longest path in the vertical constraint graph. In Chapter 2, a better theoretical lower bound is obtained.

1.5 Survey of Channel-Width Minimization Algorithms

In the last twenty-five years, much work has been done on channel routing algorithms. An excellent survey is given by Lengauer [3]. LaPaugh [4] first showed that the channel routing problem, in the Manhattan model, with no doglegs, is NP-complete. The reduction is from circular arc graph coloring. Syzmanski [5] proved that the channel routing problem in the Manhattan model, with doglegs allowed at any column, is NP-complete. His reduction is from 3-satisfiability. In addition, Sarrafzadeh [6] showed that channel routing under the knock-knee model is also NP-complete. As such, most of the algorithms are heuristics, that, nevertheless, give optimal results in most practical cases. Generally, the algorithms are applicable to both VLSI circuits and PCBs. However, some algorithms are specialized; for example, the recent 'over-the-cell' routing algorithms [7],[8] are applicable only to VLSI circuits. Multilayer algorithms for five or more layers are targeted towards PCB applications or future VLSI circuits.

1.5.1 Two-layer Algorithms

The first channel routing algorithm was developed by Hashimoto and Stevens [1]. They ignored the vertical constraint graph and sorted the nets by their left edges. A pass was then made through the sorted list and as many nets as possible were placed on the first track. The placed nets were then deleted from the sorted list. A pass through the remaining nets was made to fill the second track. This was continued until no more nets were left. Since they ignored the vertical constraints, the routing caused short circuits among some of the vertical constraints. These conflicts were resolved by

assuming that intermediate columns existed that would allow the overlapped vertical segments to be separated, which was a reasonable assumption for low-density PCBs. This is the famous 'left edge algorithm' (*lea*) which is the basis for many heuristic algorithms. Since sorting was involved it has a time complexity of $\mathcal{O}(N \lg N)$, where N is the number of nets. A much more efficient procedure was presented in Fig. 1.8, with a time complexity of $\Theta(N)$. Deutsch [9] used doglegging to minimize the number of tracks needed. His algorithm was also based on the *lea* with the variation that tracks were alternately assigned from top and bottom and the nets were placed from the left as well as from the right. Yoshimura and Kuh [10] proposed two algorithms which minimized the longest path in the vertical constraint graph. The first algorithm attempts to do this by merging nodes in the vertical constraint graph, whereas, the second algorithm used bipartite matching to do the same. Rivest and Fiduccia [11] proposed a *greedy* algorithm which proceeds column by column and uses a set of heuristic rules to assign the tracks at a column. The hierarchical algorithm proposed by Burstein and Pelavin [12] uses a divide and conquer method, and is one of the most effective routing algorithms available.

Only a few exact (optimal) algorithms are available. Kernighian, Schweikert and Persky [13] used the branch-and-bound method, whereas, Wang and Lee [14], and Lin [15] used the A* search method. These methods are not of practical interest because they often take too much time. For example, in one trial, a problem instance took 4.5 hours with the Kernighian, Schweikert and Persky's branch-and-bound method, whereas, the same problem was solved to optimality in less than a second by Yoshimura and Kuh's algorithm.

1.5.2 Multilayer Algorithms

With the advent of CMOS technology, multilayer routers are being used by practitioners of CMOS routing. Generally, two metal layers and a polysilicon layer is used for the routing. Many multilayer algorithms merely extend the single-direction-per-layer paradigm of the Manhattan model. For example, in the vertical-horizontal-vertical (VHV) model, the first layer (V) is used to route the vertical segments that are connected to the upper side of the channel; the second layer (H) is used for the horizontal segments of the nets, and, the third layer (V) is used for routing the vertical segments that connect to the lower side of the channel. Consequently, vertical segments never short each other, (but they do overlap) and, therefore, there are no vertical constraints, which allows the channel to be routed in density. Many practitioners prefer the HVH model in the hope of reducing the width of the channel by half. In this model, the vertical constraints again become relevant. Pitchumani and Zhang [16] suggests that some mixture of VHV and HVH may be best. Chen and Liu [17] extended the algorithm of Yoshimura and Kuh [10] to an HVH model. Bruell and Sun [18] modified the column-by-column heuristics of Rivest and Fiduccia [11]. Heyns [19] used the left-edge algorithm with doglegs but departed from the strict one-direction-per-layer paradigm. The algorithm allowed both horizontal and vertical segments at the top layer, horizontal segments only on the middle layer and vertical segments only on the bottom layer. Instead of extending a two-layer routing algorithm, Cong et. al [20] started with an actual two-layer routing and transformed it into an HVH routing by placing two tracks into a single two-layer track, provided the vertical segments do not short. Enbody and Du [21] extended the HVH model to HVHV,...,H model and also to the VHV,...,V model. Braun et. al [22] took a different approach. They divided the layers into sets of two layers and three layers and distributed the

nets among these sets. They, therefore, reduced the multilayer problem to several two-layer and three-layer routing problems.

Chapter 2

Application of Neural Networks to Channel-Width Minimization

Artificial neural networks are derived from consideration of biological *neurons* and their interconnections, which can solve problems in the areas of classification, pattern matching, pattern completion, noise removal, optimization, and control that no available digital computer can do adequately. Modern digital computers are good at well-defined tasks amenable to algorithmic descriptions. The artificial neural network is an attempt to design computational systems with brain-like capabilities. Artificial neural networks have been studied for almost fifty years, but the recent resurgence of interest in neural networks is largely due to the easy availability of superior computer systems on which they can be simulated, improved VLSI implementation technologies and better-understood learning algorithms and improved theoretical foundations. Neural networks consist of simple processing elements (neurons) and interconnections between them. They are also known as *connectionist models* because the solution of the problem is in the interconnections. The key to the success of artificial neural

networks is that they provide a computational model for a massively parallel ultra-fine-grained computer, and, an added advantage is that they provide a greater degree of robustness or fault tolerance compared to von Neumann sequential computers because there are many more processing nodes, each with primarily local connections. Damage to a few nodes or links thus need not impair overall performance significantly. Most neural net algorithms also adapt connection weights in time to improve performance based on current results. Their use has been dramatically successful in the area of *pattern recognition* and to a much lesser extent in solving *constrained optimization* problems. There are many models but they can be classified into two types: *feedforward* and *feedback*. In the *feedforward* networks, the first layer of neurons receive the inputs and then distributes its outputs to the next layer of neurons, and this continues in a forward direction until the output layer is reached. In the *feedback* networks, the output of a neuron may be fed back into the network as an input to other neurons. In addition, a neuron may have an external stimulus. *Feedback* networks are also called *recurrent* networks. Such networks have been used for unsupervised learning [23], self-organization [24], retrieving stored memory patterns [25], and computing solutions to a variety of optimization problems [27], [28], [29], [30], [31],[32]. Because of feedback, the outputs of these networks must stabilize, before they are interpreted, which often fails to do so. Since this thesis deals with *optimization*, as applied to the *channel routing* problem, only networks that allow optimization problems to be solved are discussed.

2.1 The Artificial Neuron

The artificial *neuron* is a simple processing element, that sums its weighted inputs, which is then mapped into its output by a nonlinear *activation* function. A conceptual

sketch of a neuron is shown in Fig. 2.1, and some typical activation functions are shown in Fig. 2.2. Note that each neuron i has an *internal* state or *activation level* U_i, and the output of the neuron, V_i, is simply

$$V_i = \mathcal{G}(U_i) \tag{2.1}$$

where, $\mathcal{G}$ is the activation function, which is typically a sigmoid function.

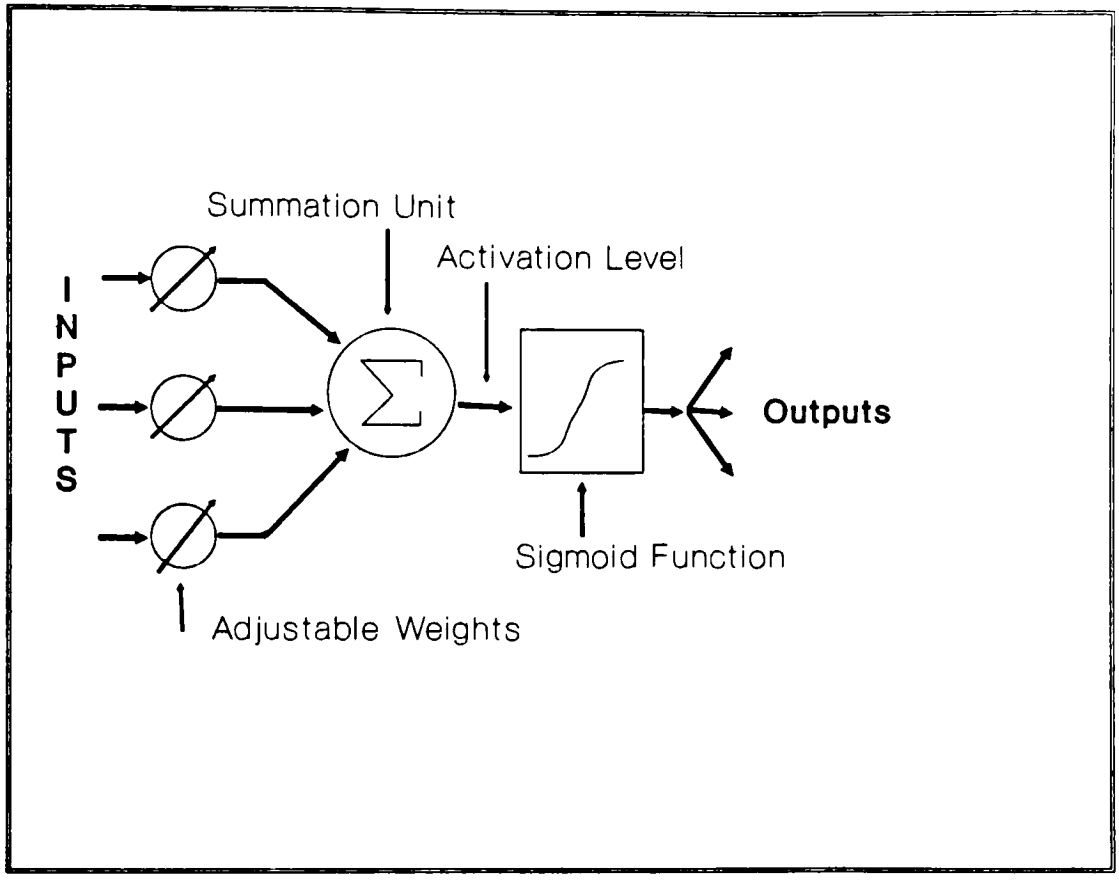


Figure 2.1: Sketch of an artificial neuron model [31]. The inputs are from other neurons and also possibly from an external source. The adjustable multiplicative weights correspond to biological synapses. Positive weights are normally used for excitory connections and negative weights for inhibitory connections. The weighted inputs are accumulated and then passed to an activation function which determines the neurons response, which, depending on the model used could be continuous or discrete.

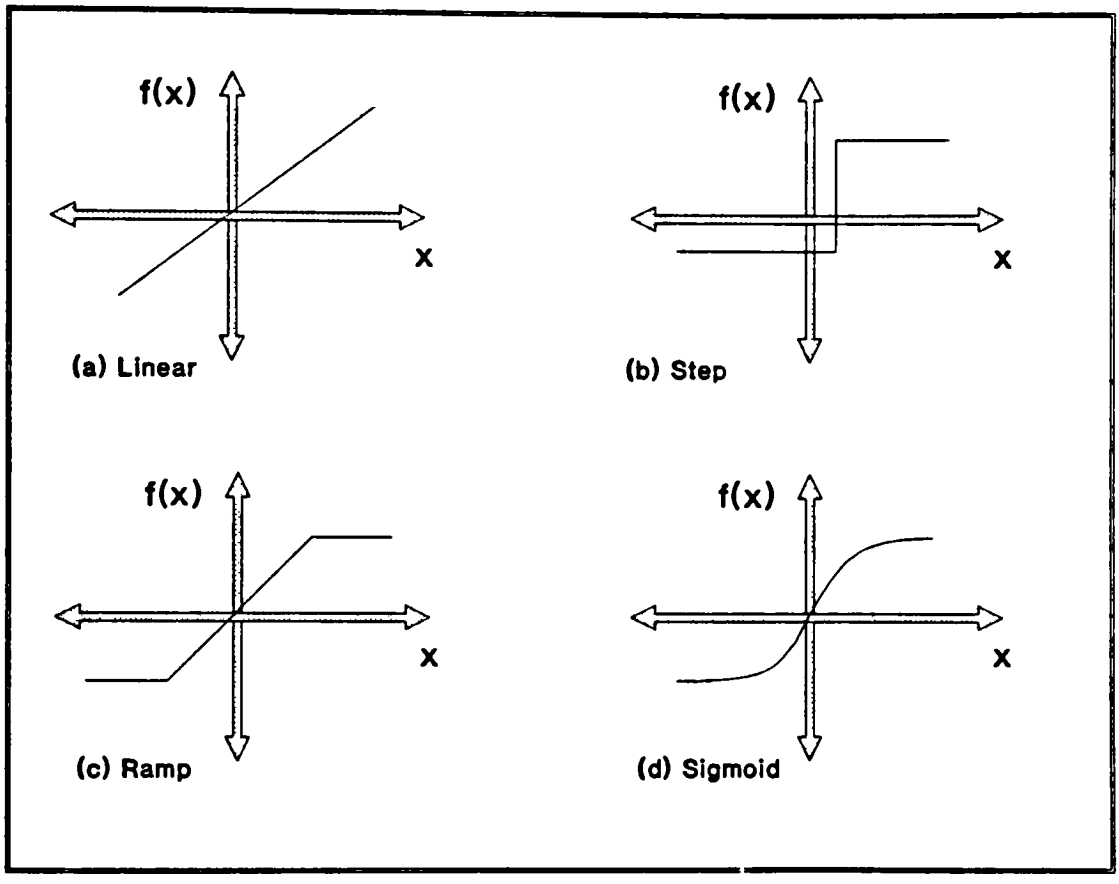


Figure 2.2: Some common activation functions. A widely used sigmoid function is the logistic function $1/(1+e^{\alpha x})$. Other sigmoid functions include the hyperbolic tangent function $\tanh(x)$ and the augmented ratio of squares $x^2/(1+x^2)$. Not shown is the Gaussian function which is also common.

2.2 The Discrete Hopfield Network

The discrete Hopfield network [25], [26] is a fully-connected, iterative, autoassociative net primarily used as associative memory but can also be used for optimization as shown by experiments described later in this Chapter. The weights are symmetric with no self-connection. Each neuron can receive inputs from all other neurons and, in addition, can receive an external input. For a Hopfield network to work properly, the following conditions must be satisfied:

- The interconnection (synaptic) weight matrix must be *symmetric*.
- There must no self-loops; in other words, the diagonal weight matrix elements must be zero.
- Only one unit should update its activation at a time.
- Each unit continues to receive an external input.
- The inputs to each unit or the output of each unit could be either binary (0,1) or bipolar (-1,+1).
- Each unit i, adds up the weighted inputs and the external input $\mathcal{I}_i$ to obtain its internal activation level $U_i = \sum_j V_j W_{ji} + \mathcal{I}_i$, which is then compared with the threshold Γ_i to yield the final output. Thus,

$$V_i = \begin{cases} 1 & \text{if } U_i > \Gamma_i \\ V_i & \text{if } U_i = \Gamma_i \\ 0 & \text{if } U_i < \Gamma_i \end{cases} \tag{2.2}$$

The asynchronous updating of the units allows a function, known as the *energy*, which is a Lyapunov function, to be found for the net. The existence of such a function

guarantees that the net will converge. This *energy* E is given by

$$E = -\frac{1}{2}\sum_i V_i \left(\sum_{j \neq i} W_{ij} V_j + \mathcal{I}_i - \Gamma_i\right) \tag{2.3}$$

Hopfield proved that the net described above will converge to a stable state if the energy function is a function that is bounded below and is a non-increasing function of the state of the system. For a neural net, the state of the system is the vector of neuron outputs. So, on iteration, since the energy is bounded and it cannot increase when a neuron state is changed, the energy has to decrease or reach a stable equilibrium.

2.3 The Continuous Hopfield Network

As mentioned in the previous section, Hopfield's first model employed two-state neurons. He next introduced [29] a modified version of his earlier model, which employed a continuous non- linear activation function to describe the output behaviour of the the neurons. He also showed that such a model could be used to solve difficult constrained optimization problems. The model consists of N interacting neurons, governed by a set of coupled nonlinear differential equations. The time dependent equation for the internal activation level U_i of neuron i may be expressed by the equation of motion

$$C_i \frac{dU_i}{dt} = \sum_{j=1}^{N} W_{ij} V_j - \frac{U_i}{R_i} + \mathcal{I}_i, \tag{2.4}$$

where,

$$V_j = \mathcal{G}_j(U_j). \tag{2.5}$$

Here, R_i is the parallel combination of the input resistance and the resistance used to model synaptic connectivity:

$$\frac{1}{R_i} = \frac{1}{\rho} + \sum_{j=1}^{N} \frac{1}{R_{ij}}. \tag{2.6}$$

If for simplicity we assume constant values for the R_i and C_i, and if all the sigmoid functions $\mathcal{G}_j$ are identical, then the equations of motion become

$$\frac{dU_i}{dt} = -\frac{U_i}{\tau} + \sum_{j=1}^{N} W_{ij} V_j + \mathcal{I}_i, \tag{2.7}$$

where $\tau = RC$ and $V_i = \mathcal{G}(U_j)$. For convenience, W_{ij}/C and $\mathcal{I}_i/C$ has been redefined as W_{ij} and $\mathcal{I}_i$ respectively. The term $-\frac{U_i}{\tau}$ is a passive decay term that causes U_i to decay toward 0, at a rate proportional to τ. The output V_i corresponds to the mean firing rate of a biological neuron. It has been pointed out by several authors [32] that this decay term slows down the convergence of a Hopfield network. For the activation function $\mathcal{G}(U)$, Hopfield suggested the following sigmoid function:

$$\mathcal{G}(U) = 0.5[1 + \tanh(\lambda \cdot U)], \tag{2.8}$$

where, λ is the gain of the sigmoid function $\mathcal{G}$. The dependence of $\mathcal{G}$ on λ is shown in Fig. 2.3.

It is not difficult to show that for a dynamical system, as described above, the time evolution of the neuron states is such that the energy function as given in Eq. (2.3) is minimized provided the following conditions are met:

1. The elements of the weight matrix W_{ij} are symmetric and the diagonal elements are zero.

2. The activation function $\mathcal{G}(U_i)$ is nondecreasing.

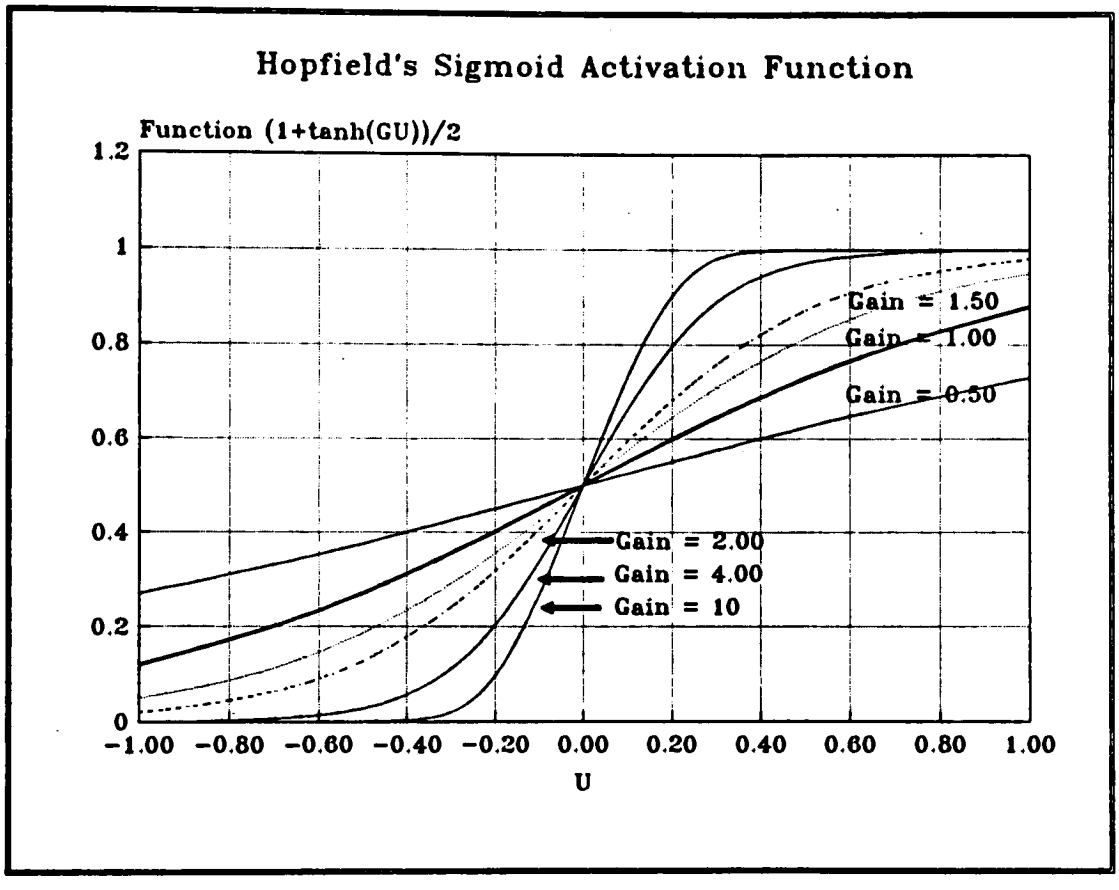


Figure 2.3: The hyperbolic tangent sigmoid used by Hopfield. The output ranges between 0 and 1. However, for large values of the gain parameter G, the output switches from 0 to 1 very quickly with respect to the input at $U = 0$. During simulation, the gain is gradually increased, thus "sharpening" the activation function.

2.4 The Boltzmann Machine

The Boltzmann machine introduced by Hinton and Sejnowski [33] combines interesting properties from both neural computing and simulated annealing [34], resulting in a powerful computational model exploiting parallelism in a natural way. It can be viewed as a generalization of the Hopfield's content-addressable memory. It was pointed out in previous sections that the Hopfield model gets trapped in local minimas. The Boltzmann machine provides a means for escaping from such a local minima by adopting a stochastic state transition compared to the deterministic update rule used by the Hopfield model. In the Boltzmann machine, it is customary to maximize a quantity called the *consensus* instead of minimizing the *energy* as described in the previous sections. The two approaches have been shown to be identical [34]. The *consensus* $C(k)$ in the configuration k, is defined as the sums of the strengths of the activated connections, i.e.

$$C(k) = \sum_{i=1}^{N} V_i(k) \left[\sum_{j=i+1}^{N} W_{ij} V_j(k) + \mathcal{I}_i \right]. \tag{2.9}$$

As remarked in previous sections, W_{ij} are the symmetric synaptic strength weight matrix elements. $V_i(k)$ is the state of unit i in configuration k. The consensus is large if many excitory connections are activated, and it is small if many inhibitory connections are activated. The consensus, therefore, is a global measure indicating to what extent the units in a Boltzmann machine have reached a consensus about their individual states, subject to the desirabilities expressed by the individual connection strengths. The consensus maximization process in a Boltzmann machine depends on how the updates are made. The following two models and their submodels are common:

1. **Sequential Boltzmann Machines** Units are allowed to change their states *only* one at a time.

2. **Parallel Boltzmann Machines** Units are allowed to changes their states simultaneously. These are categorized into two sub-categories:

   Synchronous Parallelism In this scheme sets of state transitions are scheduled in successive trials, each trial consisting of a number of individual state transitions. After each trial, the accepted state transitions are communicated through the network so that all units have up-to-date information about the states of their neighbors before the next trial is initiated. During each trial, each unit is allowed to propose a state transition exactly once. Evidently, synchronous parallelism requires a global clocking scheme to control synchronization. The following two cases are distinguished:

   - *Limited Parallelism:* Units may change their states in parallel only if they are not adjacent.
   - *Unlimited Parallelism:* Units may change their states in parallel whether or not they are adjacent. Clearly, erroneous states can be generated. Proving the asymptotic convergence of the Boltzmann machine for this model is still an open problem.

   Asynchronous Parallelism State transitions are simultaneously and independently proposed and then accepted or rejected. The information used for such an evaluation is not necessarily up-to-date. Since it does not require a global clock, asynchronous parallelism is normally easier to implement in hardware. Again the two cases of *limited* and *unlimited* parallelism need to be considered separately.

2.4.1 Sequential Boltzmann Machine

Let a Boltzmann machine be in configuration k, then a *neighboring configuration* k_j is defined as the configuration that is obtained by complementing the state of unit j. The difference in consensus $\Delta\mathcal{C}(k,j)$, when the state of unit j is changed to j', in configuration k, is

$$\begin{aligned}\Delta\mathcal{C}(k,j) &= \mathcal{C}(k,j') - \mathcal{C}(k,j) \\ &= (\alpha - 2V_j(k))\left[\sum_{i=1}^{N} W_{ij}V_i(k) + \mathcal{I}_j\right] \end{aligned} \tag{2.10}$$

where, $\alpha = 1$, if binary-valued {0,1} states are used and 0 if bipolar-valued {-1,+1} states are used. It is obvious from the above equation that the change in consensus, resulting from a change in the unit j, is completely determined by the states of the neighbors of j and the corresponding connection strengths. Consequently, each unit can evaluate locally its own state transition. If we allow only single unit state changes, then the consensus has a local maxima in a certain configuration if neighboring configurations each have a lower consensus.

In a sequential Boltzmann machine, a unit j is selected, and the neighboring state, obtained by complementing the state of unit j, is accepted with a probability $\mathcal{A}$. Following the ideas of simulated annealing, the acceptance probability $\mathcal{A}$ is a function of the difference in consensus $\Delta\mathcal{C}(k,j)$ and a control parameter T and is given by the well-known Boltzmann distribution

$$\mathcal{A}(k,j,T) = \frac{1}{1+\exp\left(\frac{-\Delta\mathcal{C}(k,j)}{T}\right)}, \tag{2.11}$$

where, $\Delta\mathcal{C}(k,j)$ is the change in consensus when unit j changes state in configuration k, and is given by Eq. (2.10). The acceptance probability function $\mathcal{A}(k,j,T)$ is shown in Fig. 2.4 as a function of $\Delta\mathcal{C}(k,j)$ for several different values of the control param-

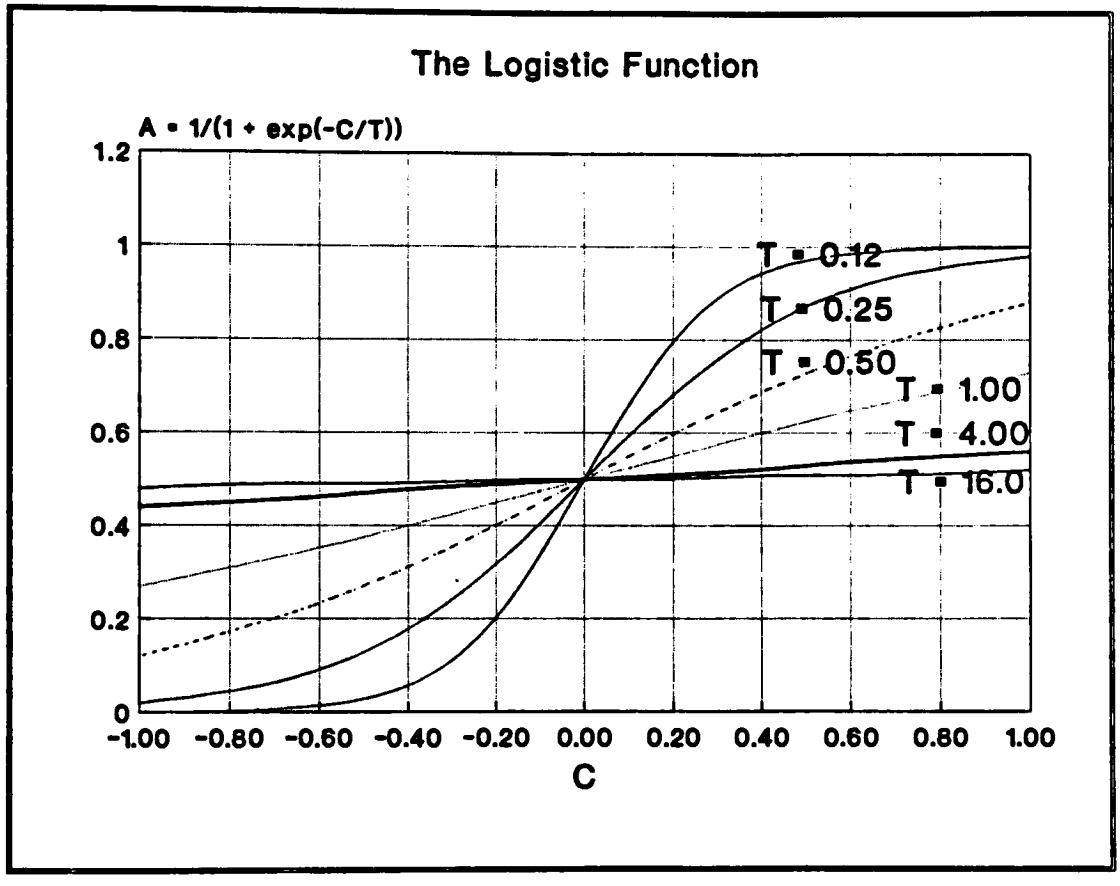


Figure 2.4: The logistic acceptance probability function $\mathcal{A}$ normally used in a *Boltzmann* machine. The acceptance probability function $\mathcal{A}$ changes slowly for $\mid C \mid > 1$. For small values of T, the probabilty function changes from 0 to 1 very quickly in the neighborhood of $C = 0$.

eter T. It can be shown [34] that the Boltzmann machine converges asymptotically to the set of globally optimal configurations. In practice, however, it will converge to a (near-)optimal one, because of the finite-time approximation. To obtain a solution, the Boltzmann machine is started with a sufficiently large value of T and a randomly chosen initial configuration. Subsequently, for each value of T, the machine is run until the consensus does not increase appreciably, at which point, T is decremented and the process repeated. As T approaches 0, state transitions become more and more infrequent, and finally the Boltzmann machine stabilizes in a locally maximal configuration which is expected to be a near-optimal configuration. This process of running the machine at a fixed temperature, until it stabilizes and then lowering the temperature and running the machine again is borrowed from *simulated annealing*. Lowering the *control parameter* (temperature) too fast will not yield an optimal solution and lowering too slowly takes too much time. Before a Boltzmann machine is run, a cooling schedule is determined. This is usually of the form

$$T = \frac{\mathcal{D}}{\log(1+t)}, \tag{2.12}$$

where, $\mathcal{D}$ is the depth of the deepest local minima and t is a count of the number of times T was decremented. However, a practical method is to multiply T by a constant β slightly less than 1, typically 0.95. The initial value of T is normally chosen such that the acceptance probability is about 0.5.

A typical algorithm for the Boltzmann machine is given in Fig. 2.5.

```
(* Boltzmann Machine.
        NEpochs = Number of epochs per temperature
        NbrOfNeurons = number of units
Step 1: Initialize weights, T and configuration
Step 2: Randomly select a unit j
Step 3: Compute Change in Consensus as given by Eq. (2.10)
Step 4: Accept the change in state of unit j with probability
        given by Eq. (2.11)
Step 5: Repeat Steps 2,3,4 until stop criterion is met
*)

   Initialize Weights
   Initialize T
   Randomly assign the initial configuration
   REPEAT
        NChanges := 0
        FOR k := 1 TO NEpochs * NbrOfNeurons DO
           BEGIN
              Randomly Select a unit j
              Compute Change in Consensus DeltaC {Eq. (2.10)}
              Compute the Acceptance Probability A {Eq. (2.11)}
              Generate a random number R between 0 and 1
              IF R < A THEN
                 BEGIN
                    Complement State of unit j
                    NChanges := NChanges + 1
                 END
           END
       T := Beta * T
   UNTIL T <= StoppingTemp OR NChanges = 0
```

Figure 2.5: A typical Boltzmann machine simulation algorithm. The stopping criteria varies with the application.

2.5 Previous Applications of Neural Networks to the Channel-Width Minimization Problem

Many sequential algorithms have been proposed for channel routing, but only a few massively parallel algorithms using neural networks have been proposed. Out of the proposed neural network algorithms [35],[36],[37],[38],[39], only two seem viable and both have been implemented and studied in this thesis. The one proposed by Funabiki and Takefuji [35], based on a maximum net and for multilayers appears to work reasonably well after a suitable modification, but the other network proposed by Shih and Feng [36] for two layers has severe difficulty in finding solutions for even very small (three nets) problem instances. It should be remarked here that finding a routing on four layers is easier than finding a solution in two layers, because the vertical constraint graph is split and there are, therefore, more solutions available and the nets are easier to place.

In each of the network architectures to be discussed below, the total number of neurons required is equal to the product of the number of tracks, (which is provided), and the number of nets. The two-layer solution of our example problem, using a neural network, is shown in Fig. 2.6. Each neuron is identified by a pair of subscripts, one for the net and the other for the track. If a neuron with subscripts i and j is on, it signifies that the solution has assigned net i to track j. In the multilayer model used here, the routing region consists of one or more pairs of layers, with each pair similar to the conventional two-layer pair. Each layer-pair is assumed to be independent of the other layer-pairs. It is also assumed that each terminal is available in every layer-pair. Each neuron for the the four-layer case is indexed with three subscripts. If neuron ijk is on, it means that net i has been assigned to track j in the layer-pair

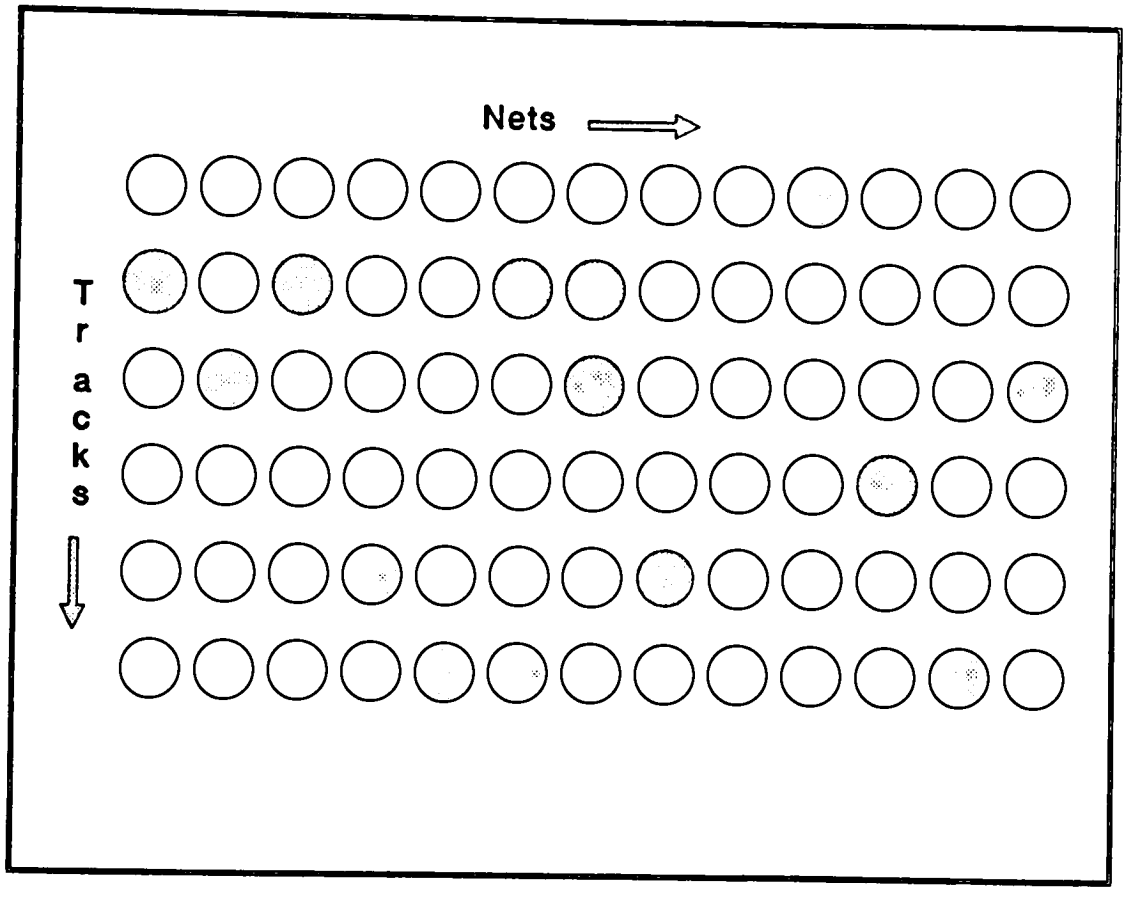


Figure 2.6: The neural network solution of the problem instance described in Fig. 1.2. The shaded neurons represent neurons that are on. Each neuron is identified by two subscripts i, and j. If neuron ij is on, it means that net i has been assigned to track j. The interconnection weights are not shown since there are too many – 3003 to be exact.

k. The interconnection strengths between the neurons are primarily determined by the horizontal and vertical constraints.

2.5.1 Funabiki and Takefuji's Algorithm

Funabiki and Takefuji [35] used the McCulloch-Pitts neuron model. They claim that Hopfield's equation of motion with the passive decay term and the sigmoid transfer function is detrimental to the rate of convergence. For n nets and m tracks per layer-pair, the equation of motion proposed by them for the ijk-th neuron is given by the first-order differential equation of the time rate of change of a neuron's input U_{ijk} as:

$$\frac{dU_{ijk}}{dt} = -A\left(\sum_{q=1}^{m}\sum_{r=1}^{2} V_{iqr} - 1\right) - B(H+K) + Ch\left(\sum_{q=1}^{m}\sum_{r=1}^{2} V_{iqr}\right), \qquad (2.13)$$

where, the equation is for the i-th net assigned to the j-th track in the k-th layer-pair. The first term forces one and only one output among the $2m$ processing elements to be non-zero corresponding to the i-th net. The second term is for the inhibitory forces due to the violation of the horizontal and the vertical constraints. H is simply the number of horizontal conflicts and K the number of vertical conflicts if the proposed assignment is made. Note that the determination of H and K require that the states of locally connected neurons be known. The last term is a hill-climbing term and is expected to allow the system to escape from the local minimum and converge to the global minimum, where, $h(x)$ is 1, if $x = 0$, and 0 otherwise. A, B, and C are weighting factors. The output V_{ijk} of the ijk-th processing element is related to the input U_{ijk} by:

$$V_{ijk} = 1 \text{ if } U_{ijk} > 0 \text{ and } U_{ijk} = \max\{U_{iqr}\} \text{ for } q = 1,\ldots,m \text{ and } r = 1,2 \qquad (2.14)$$

To simulate the system, one must integrate the above differential equation, which can be approximated by the first-order *Euler* equations. They proposed a parallel synchronous update of the neurons. But experiments, reported in this thesis, showed that parallel synchronous update led to spurious states, which were not feasible (had

conflicts in assignments) even when the net seemed to have converged to the optimal solution. An asynchronous update took care of the problem. It is necessary to point out here that the algorithm does NOT find the minimum channel width. The algorithm is provided with the minimum width and then it attempts to find a legal routing that does not conflict with the horizontal and vertical constraints. The problem solved, is, therefore, a *constraint satisfaction* problem, instead of a *constrained optimization* problem. The outline of their algorithm is shown in Fig. 2.7.

The authors have reported finding four-layer solutions to seven of the standard problems from the literature, but the solution to the Deutsch *difficult* problem was not *optimal*. Their algorithm, as implemented in this thesis, could not find solutions to all seven problems and the experimental results as found in this study are reported in the next section. They suggested a two-phase algorithm, in which, in the first phase, the longer nets were to be routed and in the second phase, the remaining nets were to be routed keeping the longer nets fixed. This strategy was not implemented in the current work because no further details on this phase of the algorithm was provided by the authors. It is important to note the following about Funabiki and Takefuji's algorithm.

- The number of neurons required is $n \cdot m \cdot k$, where, n is the number of nets, m is the number of tracks per layer-pair, and k is the number of layer-pairs. Since the number of interconnecting wires grows quadratically with the number of neurons, it would be desirable to partition the neurons into separate sets, one set for each layer-pair.

- The algorithm requires two phases. In the first phase, nets, whose lengths are greater than 30 % of the total channel length, are routed. In the second phase, the remaining nets are routed, keeping the track assignments of the first set

```
t := 0;
T_max := 500;
A := B := 1; C := 10
∀ijk, U_ijk ← random values
∀ijk, V_ijk ← 0
REPEAT
    t := t + 1;
    for i := 1 to n
        for j := 1 to m
            for k := 1 to 2
            U_ijk := U_ijk + ΔU_ijk
    (* end loops i,j, k *)
    for i := 1 to n
        U_i-max = max {U_iqr } for q = 1,...,m and r = 1,2
        for j := 1 to m
            for k := 1 to 2
            if U_ijk > 0 AND U_ijk = U_i-max then
                    V_ijk = 1;
            else V_ijk = 0;
    (* end loops i,j, k *)
UNTIL t >= T_max OR no conflicts in assignment of nets
(* End Main *)
Function ΔU_ijk
    Q := -A (Σ_{q=1}^{m} Σ_{r=1}^{2} V_iqr - 1) + Ch
    if t mod 10 ≤ 5 then
        return Q - B(H + K) V_ijk
    else return Q - B(H + K)
(* End Function ΔU_ijk *)
```

Figure 2.7: Funabiki and Takefuji's multilayer channel routing algorithm. This algorithm was implemented in the present work. The model used the *maxnet* concept for each group of neurons representing a single net, and for each neuron, the McCulloch-Pitts model was assumed. However, the longer nets were not routed first as suggested by the authors, because not enough details were given to implement the suggestion. Non-optimal results were reported by the authors for the four-layer solution of the *difficult* channel.

(longer nets) fixed. They claim that this procedure is essential, otherwise, it is not possible to find a track assignment for the longer nets. It is not clear how this last step can be done on a neural network and if it can be done, what effect it has on the implementation.

- The parameters A, B, and C determine the relative contributions of the horizontal conflicts, the vertical conflicts, one net to one track constraint, and a hill-climbing term. It is not clear how these parameters are to be determined and what effect they have on convergence.

- The convergence reported by the authors is not good, and ranges from 7% to 76%, even with the two-phase algorithm.

- Non-optimal results were obtained for the *difficult channel* problem example. They needed eleven tracks, whereas, ten tracks are sufficient as shown in the next section.

- Results were reported for four and more layers. Two layer solutions, which are more difficult to obtain were not reported.

2.5.2 Shih and Feng's Algorithm

Shih and Feng [36] used the Hopfield and Tank's model [29], and proposed an algorithm for the two-layer problem, which can, however, be easily extended to four layers. The equation of motion for neuron i, in the Hopfield and Tank model is given by:

$$C_i(dU_i/dt) = \sum_j W_{ij}V_j - U_i/R_i + \mathcal{I}_i, \quad (2.15)$$

$$V_i = g_i(U_i). \quad (2.16)$$

In the above equation, U_i is the input to neuron i, and V_i is the output of neuron i, related by the gain function g_i. C_i is the input capacitance of neuron i, and R_i is the input resistance. $\mathcal{I}_i$ is an external bias and W_{ij} is the synapse (interconnection) strength between neurons i and j. For the channel routing problem in two layers, the neurons are arranged in a two-dimensional array, each neuron having two subscripts, the first refers to the net and the second to the track. The synapse strength, $W_{ij,kl}$, is the interconnection weight between neurons ij and kl. Neuron ij is on if net i is assigned to track j. Similarly, neuron kl is on if net k is assigned to track l. The connection strengths are again determined by the vertical and horizontal constraints. The proposed sum over the connection strengths is given by

$$\sum_{kl} W_{ij,kl}V_{kl} = -A(\sum_{l=1}^{m} V_{il} - V_{ij}) - B\sum_{k=1}^{n} \mathrm{hcg}(k,i)V_{kj} - C\sum_{k=1}^{n}\sum_{l=1}^{j} \mathrm{vcg}(i,k)V_{kl}, \quad (2.17)$$

where, the first term ensures that at least one track is assigned to each net and the expression inside the parenthesis is n, if $n+1$ tracks has been assigned to net i. The second term is to inhibit horizontal conflicts and the third term to inhibit vertical conflicts. The function $\mathrm{vcg}(i,k) = 1$, if k is a descendant of i in the vertical constraint graph, which simply means that net k should be assigned a track below

net i. Similarly, the function $\text{hcg}(i,k) = 1$, if net i overlaps net k. If m tracks are available for routing, then the external bias I_{ij}, was taken as:

$$I_{ij} = D \cdot m(1 - 11 \times \text{outrange}(i,j)). \tag{2.18}$$

In Eqs. (2.17) and (2.18), A, B, C and D are parameters that affect convergence of the solution. The authors suggest the following values: $A = 100; B = 30; D = 7$. These were determined after a difficult heuristic search whose details were not elaborated by the authors. The function $\text{outrange}(i,j) = 1$, if assigning net i to track j violates the range of tracks possible for net i as determined from the *vertical constraint* graph. This may be stated as:

$$\text{outrange}\ (i,j) = \begin{cases} 1 & \text{if} \quad j > m - \mathcal{L}_{\max} + \mathcal{L}_i \quad \text{or} \quad j < \mathcal{L}_i \\ 0 & \text{otherwise} \end{cases} \tag{2.19}$$

where, $\mathcal{L}_i$ is the level of net i in the vertical constraint graph and $\mathcal{L}_{\max}$ is the maximum level in the vertical constraint graph. It should be noted that the tracks allowed for a net is *overly restricted* and this may prevent the network from finding a solution. To correct the *mistake*, $\mathcal{L}_{\max}$ should be interpreted as the maximum level of a chain to which net i belongs and not the maximum level of the entire vertical constraint graph.

They suggest a parallel synchronous update of the neuron states. The transfer function g was taken as:

$$g(U_{ij}) = \arctan(U_{ij} \cdot G)/\pi + 0.5, \tag{2.20}$$

where, G is the gain of the neurons, which was initialized to 0.8 and multiplied by 1.01 at every iteration. This was expected to help the system climb out of a local minima and converge to a more stable state. The proposed algorithm was implemented in this

Assume m to be the number of tracks available
$\forall ij,\ U_{ij} \leftarrow 0$
$\forall ij, V_{ij} \leftarrow 1/m$
set parameter values
repeat
    for each neuron ij
    begin
        $\mathrm{d}U_{ij} = (\sum_{kl} W_{ij,kl} V_{kl} - U_{ij}/R_{ij} + I_{ij})\mathrm{d}t/C_{ij}$
        $U_{ij} = U_{ij} + \mathrm{d}U_{ij}$
    end
    for each neuron ij
        $V_{ij} = g(U_{ij})$
until $|\ \mathrm{d}V_{ij}\ | < \epsilon, \forall ij$

Figure 2.8: Shih and Feng's algorithm for the two-layer channel routing problem. It is based on the continuous Hopfield network. The algorithm was implemented in the present work, but did not give feasible solutions. Eq. (2.17) shows how $\sum_{kl} W_{ij,kl} V_{kl}$ should be calculated, and, Eq. (2.18) shows how the external input I_{ij} is to be calculated.

study, but the network, when run, converged to an infeasible (one having conflicts) solution, again and again. The problem instances tried were small (number of nets varied from three to ten), and, yet the network repeatedly failed to converge to a feasible solution. The authors present solutions of several simple (3 to 10 nets) problems and one 21-net problem. The algorithm proposed by the authors is quite simple and is outlined in Fig. 2.8. The convergence of the network is tested with a very small constant ϵ, which was taken to be 10^{-6}. The time constant $R_{ij}C_{ij}$ was set to 1, but separate values of the resistance and capacitance were not given, as required by the program above. The time step $\mathrm{d}t$ was taken to be 10^{-3}.

2.6 Efficient Solutions to the Channel-Width Minimization Problem Using Neural Networks

Shih and Feng [36], as mentioned in the previous section, attempted to use the constraint on the track assignment imposed by the vertical constraint graph, whereas, Funabiki and Takefuji [35] did not take advantage of the vertical constraint graph to limit the selection of tracks for each net. An examination of Fig. 2.9 shows that each net, represented by a node, is constrained to a certain contiguous sequence of tracks, which we will call the *feasible* tracks of a net. This constraint is imposed because for each net i in the vertical constraint graph, there is a certain number p_i of nodes that precedes it and a certain number s_i of nodes that follows it in the *longest chain* C_i to which node i belongs. Therefore, if the number of tracks allowed is t, then the tracks that net i can occupy are $p_i + 1, \ldots, t - s_i$. The numbers, s_i and p_i, can be easily obtained by a variation of the depth-first search of a graph as shown in Fig. 2.10. The time complexity of such an algorithm is expected to be $\mathcal{O}(E)$, where E is the number of edges in the graph.

We can make these constraints even tighter by considering the set of predecessor nodes $\mathcal{P}_i$ and the set of successor nodes $\mathcal{S}_i$ of the node i. We then determine the channel densities $D(\mathcal{P}_i)$ and $D(\mathcal{S}_i)$ for each set of nodes, which can be done by a simple pass across the nets, and is a variation of the algorithm given in Fig. 1.8. The time complexity is $\Theta(N)$, where N is the number of nets. The number t_i^p which gives the minimum number of tracks that must precede net i and the number t_i^s which gives the minimum number of tracks that must follow net i are calculated as:

$$t_i^s = \max\,(s_i, D(\mathcal{S}_i)) \text{ and } t_i^p = \max\,(p_i, D(\mathcal{P}_i)) \tag{2.21}$$

The horizontal constraints, therefore, interact with the vertical constraints to replace

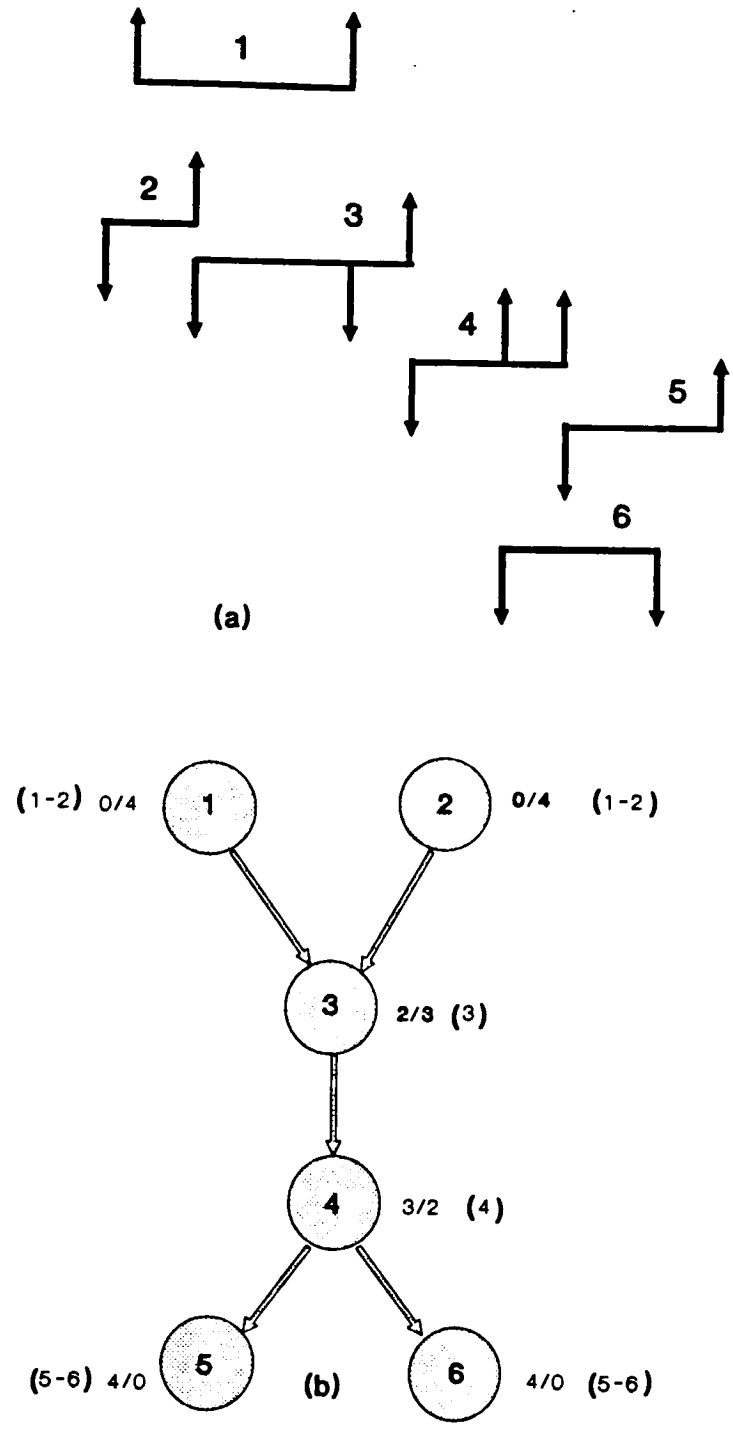


Figure 2.9: A problem instance is shown in part (a) and its vertical constraint graph in part (b). For each node, the numbers t_i^s and t_i^p, as given by Eq. (2.21), are shown next to it in the form t_i^p/t_i^s, where t_i^s is the minimum number of tracks needed by the successors of node i, and, similarly, t_i^p is the minimum number of tracks needed by the predecessors of node i. The *feasible* tracks of each node are also shown in parenthesis, assuming the nets to be routed in 6 tracks. Note that the channel density is three.

```
(* Given a directed acyclic graph in the adjacency list form,
   find for each vertex v, the number of nodes node[v].p that
   precedes it and the number of nodes node[v].s that succeeds it
   in the longest chain (path) in the vertical constraint graph
   to which it belongs.
*)

     for each vertex v do
         node[v].p := 0;
     for each vertex v with no predecessors do
         dfs_hilo(v,0)

procedure dfs_hilo(v,pred_v)
begin

     max_nodes_below := 0;
     for each vertex u adjacent to v
     begin
          if node[u].p < 1+pred_v
             dfs_hilo(u, pred_v + 1);
          if max_nodes_below < 1+ node[u].s
              max_nodes_below := 1+ node[u].s;
     end
     node[v].p := pred_v;
     node[v].s := max_nodes_below;
end
```

Figure 2.10: Variation of the depth-first algorithm which, for each node determines how many nodes precede and how many nodes follow it in the longest chain, to which it belongs, in the vertical constraint graph.

the lower bound provided by the length of the longest path in the vertical constraint graph. This new bound $\mathcal{X}_c$ is given by

$$\mathcal{X}_c = \max\{t_i^s + t_i^p + 1\}. \tag{2.22}$$

The lower theoretical bound $\mathcal{T}_c$, for the number of tracks needed for a channel is, therefore,

$$\mathcal{T}_c = \max(\mathcal{D}_c, \mathcal{X}_c), \tag{2.23}$$

where, $\mathcal{D}_c$, is the *channel density.*

Since the vertical and horizontal constraints impose constraints on the range of tracks each net may occupy, we use this information to turn off certain neurons permanently. This is done by assigning negative weights to the external (bias) input of a neuron that is to be turned off. Using this strategy alone improves the convergence rate of the neural network and reduces the number of iterations required to reach an optimal solution.

An *additional* strategy, applicable to multilayer channel routing, can significantly improve the performance of the neural network. This is based on the widely used divide-and-conquer strategy. Since each layer-pair is independent with no interactions between them, we should divide our nets into g groups, where, g is the number of layer-pairs we are attempting to route. For example, in the case of two layer-pairs (four layers in all), we would have the following significant advantages:

1. The number of neurons representing each layer-pair is roughly halved.

2. The number of interconnections is reduced by about 75%, because the number of interconnections is approximately $n^2/2$, where, n is the number of neurons.

3. The computation in each channel can proceed simultaneously since they are completely decoupled.

4. If our problem size is too big, we can choose to route only one channel at a time. Without decomposing the problem into two independent subproblems, it might not have been possible to use the network at all!

5. Problem decomposition also produces subproblems whose solutions are significantly simpler. For example, in one instance, the maximum height of the vertical constraint graph was reduced from 23 to 5. Optimal solutions are, therefore, found quickly and more frequently.

The decomposition algorithm we have used is very simple, and, produces an *optimal* decomposition with respect to *channel density.* An outline of the algorithm for the special case of a two-way split is shown in Fig. 2.11. The algorithm described in Fig. 2.11 is optimal with respect to the density of each channel, but not with respect to the longest path in the vertical constraint graph of each channel, or any other criteria. Our actual implementation of this algorithm is more complicated because we

- keep channel density optimal
- attempt to equalize the number of nets or the sum of lengths of the nets in each channel, as selected by the user.
- color nets non-deterministically

It is obvious that the partitioning of the nets should be done such that the maximum channel density among all the channels is as small as possible. Another important criteria for a good partitioning is to equally distribute the nets among the channels so as to keep the *maximum* number of neurons in a channel as small as possible. It is not clear what other criteria should be considered in the partitioning process. Intuitively,

```
(* Distribute nets into two channels (four layers) such that the
   density of each channel is optimal.
*)
       chd1 := 0;  (* Channel Density 1*)
       chd2 := 0;  (* Channel Density 2*)
 (*Scan nets from left ot right as they appear in the channel*)
       i := 1;
       while  i <= N do
          while  NetOrder[i] > 0 do
              n := NetOrder[i]
              if chd1 < chd2
                  chd1 := chd1 + 1;
                  color[n] := 1
              else
                  chd2 := chd2 + 1;
                  color[n] := 2
              i := i +1;
         end while

         while i <= N AND NetOrder[i] < 0 do
            n := NetOrder[i];
            if color[n] = 1
               chd1 := chd1 - 1;
            else
               chd2 := chd2 - 1;
            i := i + 1;
         end while

      end while
```

Figure 2.11: A simple algorithm for partitioning of nets into two channels with optimal channel density. Note that this algorithm is again derived from the algorithm in Fig. 1.8.

one may want to reduce the height of the vertical constraint graph for each channel. But, it was observed experimentally that the partitioning process is not necessarily better if the maximum height of the vertical constraint graph among all channels is as small as possible. In the majority of cases, an arbitrary partitioning appears to work because the vertical constraint graph is severely fragmented, and consequently the routing becomes easier. In cases, where $\mathcal{X}_c$ given by Eq. (2.2) exceeds the number of tracks available for the routing, we retry the partitioning process, which being non-deterministic, will most likely produce a different partition. This problem can arise if the original vertical constraint graph is dense and has many long chains whose lengths exceed the channel density. We should also remark here that Braun et. al [22], for their multilayer channel router Chameleon, also partition the nets into two or three layer sets, with the help of a three-parameter cost function. They regard the length of the longest path an important factor in the difficulty of routing.

Once the partitioning of the nets is achieved, the neural network is loaded with the interconnection weights which are primarily determined from the vertical and horizontal constraint graph of each channel. Our model uses only a few weights. They are shown in Table 2.1. In most of the experiments, the values of E, K, and C

$-E$	External Input to neurons to be kept turned off.
$+1$	External Input to neurons to be considered in the network
$-K$	Interconnection weight between neurons with horizontal or vertical constraints
$-C$	Column constraint to discourage from assigning two tracks to a single net

Table 2.1: Interconnection weights used in our neural network model. Typically we used $E = K = C = 2$.

were all set to 2.

2.6.1 Benchmark Problems

Table 2.2 shows seven benchmark problems [10] that are widely used in the literature. Several of these problems had nets emerging (or entering) from the left or the right of the channel. Such nets were tied to an imaginary column at the emerging end of the channel with a '0' terminal on the opposite side. The number of columns as shown in Table 2.2 are consequently inflated. An examination of Table 2.2 shows that all the benchmark problems can be routed in density except Deutsch's *difficult*

Problem Instance	Number of Nets	Number of Cols	Channel Density	Minimum Tracks	$\mathcal{X}_c$	$\mathcal{T}_c$	$\mathcal{L}_c$	Number of Cliques
ex1	21	41	12	12	8	12	7	8
ex3a	45	90	15	15	6	15	4	17
ex3b	47	84	17	17	11	17	9	19
ex3c	54	104	18	18	9	18	6	21
ex4b	57	119	17	17	15	17	13	31
ex5	63	130	20	20	5	20	3	33
dif	72	175	19	28	25	25	23	33

Table 2.2: Seven benchmark problems and their characteristics as determined by our programs. $\mathcal{X}_c$ is given by Eq. (2.22), $\mathcal{T}_c$ is given by Eq. (2.23) and $\mathcal{L}_c$ is the length of the longest path in the vertical constraint graph.

problem. In addition to the channel density $\mathcal{D}_c$, Table 2.2 also shows $\mathcal{X}_c$ as given by Eq. (2.22), $\mathcal{T}_c$ as given by Eq. (2.23), $\mathcal{L}_c$ the longest path in the vertical constraint graph, the number of cliques and finally the minimum number of tracks required for a "no-doglegging" routing of the problem. Table 2.3 shows another set of problem instances taken from [36]. A detailed description of these problem instances are given in Appendix A.

Problem Instance	Number of Nets	Number of Cols	Channel Density	Minimum Tracks	$\mathcal{X}_c$	$\mathcal{T}_c$	$\mathcal{L}_c$	Number of Cliques
Feng1	3	10	3	3	3	3	2	8
Feng2	4	7	3	3	2	3	2	17
Feng3	4	7	4	4	4	4	4	19
Feng4	5	10	3	4	4	4	4	21
Feng5	9	13	5	5	4	5	3	31
Feng6	10	12	5	5	4	5	4	33
Feng7	10	12	5	5	4	5	4	33
Feng8	10	12	6	6	4	6	3	33
Feng9	10	13	5	5	4	5	3	33
Feng10	13	15	7	7	5	7	5	33

Table 2.3: Problem instances taken from [10].

2.6.2 Application of the Discrete Hopfield Network to the Channel-Width Minimization Problem

By considering the simple example shown in Fig. 2.12, we will illustrate how the discrete Hopfield network can be used to solve the channel-width minimization problem. In Fig. 2.12, we show the problem instance, the vertical constraint graph and the horizontal constraint graph in parts (a), (b) and (c) respectively. For the sake of clarity, the interconnections are shown separately in Fig. 2.12(d),(e) and (f), where, each edge is inhibitory and has a weight of -2. The column constraints shown in Fig. 2.12(d) are not essential but are helpful in achieving and recognizing a solution quickly. These constraints enforce that not more than one track is assigned to a net. The external inputs (biases) are +1 for feasible tracks and -2 for infeasible tracks. These can also be regarded as interconnection weights between a dummy node, which is always on and the node itself. The interconnection edges due to the horizontal constraints, shown in Fig. 2.12 (e), are a replication of the interval graph of part (c) for each track. The edges due to the vertical constraints are shown in Fig. 2.12 (f).

Consider, for example, nets *1* and *2*. Since net *1* must precede net *2*, we must have an edge from each neuron of column 2 to a neuron of column 1 which is at the same or lower level. This ensures that if track t is assigned to net *1*, then a track less than t will *not* be assigned to net *2*.

The discrete Hopfield model was described in Chapter 2. Essentially, each neuron will sum its inputs and if the sum is greater than zero, the neuron will turn **on**, otherwise, it will turn **off**. The network uses a greedy approach trying to increase its *consensus*, which is the degree of agreement between the neurons and was defined in Eq. (2.9). So, in order to implement the discrete neural network, we should randomly consider a neuron, sum its input and if the sum is greater than zero, turn the neuron **on**, otherwise, turn it **off**, which as explained earlier will always increase the consensus. An attempted update of all the neurons is called an *epoch*. If only one neuron is updated at a time, since every neuron-state update increases the consensus, the consensus cannot keep on increasing, so, in a few epochs, the system will reach equilibrium. In fact, if we start out with all the neurons in the zero state, it will always take exactly one epoch to reach the equilibrium state, provided the weights shown in Fig. 2.12 are selected. Note that, because of the weights we have chosen, a neuron can only turn **on** if all its neighbors are **off**. Each such equilibrium state is a local maxima of the consensus (or the local minima of the energy). The goal, however, is to find the global maxima, which we will attempt to find in the next three sections. To illustrate what we have described, let us consider Fig. 2.12g. Assume that all the neurons are in the **off** state. We then randomly pick a neuron to update. If it is a neuron with a -2 external input, corresponding to an infeasible track, it will continue to be in the **off** state because the sum of its inputs will never be greater than zero. For a neuron, with a +1 external input, the sum of its inputs can be greater than zero if none of

the *adjacent* neurons are in the on state because that would contribute a -2 to the sum making it less than zero. (Two neurons are said to be *adjacent* if there is an edge between them.) We could have started with a random state, in which case, it usually takes 2-3 epochs to reach equilibrium. In a more complicated case, the equlibrium state will route a high percentage of the nets without conflicts and leave the rest of the nets unassigned. Because of the choice of weights we have made, the consensus will be maximum and exactly equal to the number of nets when a solution is found, since any conflict in the routing will decrease the consensus. A simple algorithm for the discrete Hopfield network is shown in Fig. 2.13. Our actual implementation is a more efficient version of this algorithm, which is easily obtained if we do not allow more than one track to be assigned to a net and by recognizing that all we need to know about the input is whether, or not, there is a conflict. In addition, we can explicitly restrict the selection of tracks to feasible tracks only. Fig. 2.14 shows how the sum of the neuron inputs are obtained. Again the algorithm can be made more efficient as pointed out earlier. From this small example, it is quite clear, that taking into account the feasibility of tracks can simplify finding the solution. In Table 2.4 we show the feasible tracks for each net for Deutsch's difficult problem in two layers and in Table 2.5, we show the feasible tracks for four layers. Note that in the case of four layers, the feasible tracks are not unique because it depends on the partition of the nets. We finally show, in Table 2.6, some interesting results of our experiments with the Hopfield network for the benchmark problems mentioned in Section 2.6.1. The results show that the Hopfield network will equilibrate and route a very high percentage of the nets in less than four epochs. We should also remark here that the channel-width minimization problem, as formulated here, is exactly the same as the maximal independent set problem, with the additional simplification that the

Single Channel							
Net	Feasible Tracks	Net	Feasible Tracks	Net	Feasible Tracks	Net	Feasible Tracks
1	15-22	19	13-22	37	1-16	55	21-26
2	8-12	20	20-23	38	13-27	56	10-19
3	7-11	21	2-26	39	12-20	57	21-27
4	18-22	22	1- 9	40	15-28	58	15-28
5	17-22	23	13-27	41	13-18	59	13-20
6	14-19	24	9-14	42	22-28	60	3-27
7	2- 7	25	5-12	43	17-28	61	1-26
8	4- 9	26	1-28	44	16-27	62	15-26
9	3- 8	27	6-13	45	11-16	63	10-27
10	18-28	28	21-28	46	10-15	64	2-26
11	19-22	29	5-10	47	11-20	65	2-27
12	2- 9	30	12-17	48	2-11	66	11-28
13	16-21	31	3-12	49	14-28	67	1-26
14	15-20	32	9-13	50	14-20	68	22-27
15	1- 6	33	1- 7	51	2-20	69	23-28
16	3-10	34	15-28	52	1-17	70	1-25
17	16-28	35	5-10	53	14-27	71	4-28
18	1- 8	36	1- 8	54	21-27	72	22-28

Table 2.4: Feasible tracks for the Deutsch's difficult problem in two layers. The number of tracks required is assumed to be 28.

cardinality of the solution set is known. To the best of our knowledge this observation has not been made previously. Since the maximal independent set problem has been widely studied and many interesting heuristics exist, it may be interesting to compare results obtained from such heuristics with those obtained in this work. The maximal independent set problem can also be easily formulated as an integer linear programming problem.

Channel 1				Channel 2			
Net	Feasible Tracks	Net	Feasible Tracks	Net	Feasible Tracks	Net	Feasible Tracks
1	6- 8	44	1- 9	2	1-4	31	2- 3
3	4-10	45	1-10	5	2-9	32	2- 4
4	1- 8	48	2-10	7	2-10	33	1- 2
6	5- 7	49	2-10	10	4-10	38	6- 9
8	2- 4	50	2- 7	12	2- 9	39	6- 7
9	1- 3	52	1- 8	14	1- 8	40	7-10
11	7- 9	53	1-10	15	1- 9	42	2-10
13	1-10	56	1-10	16	3-10	46	6-10
17	1-10	59	1- 6	18	1- 7	47	1- 6
20	7-10	60	3- 9	19	6-10	51	1-10
23	1- 9	61	1- 8	21	2-10	54	1-10
29	3- 9	63	2- 9	22	1- 3	55	6- 9
34	6-10	64	1- 8	24	5- 5	57	1- 9
35	3- 5	66	4-10	25	1- 3	58	2-10
36	1- 3	67	1- 9	26	1-10	62	1-10
37	1-10	69	1-10	27	3- 4	65	1- 9
41	1- 6	71	4-10	28	3-10	68	6-10
43	2-10	72	1-10	30	6-10	70	1-10

Table 2.5: Feasible tracks for the Deutsch's difficult problem for four layers. Since the channel density of the original problem is 19, an optimal split will leave each channel with a density of 10 or less. Ten tracks are assumed to be required for each channel.

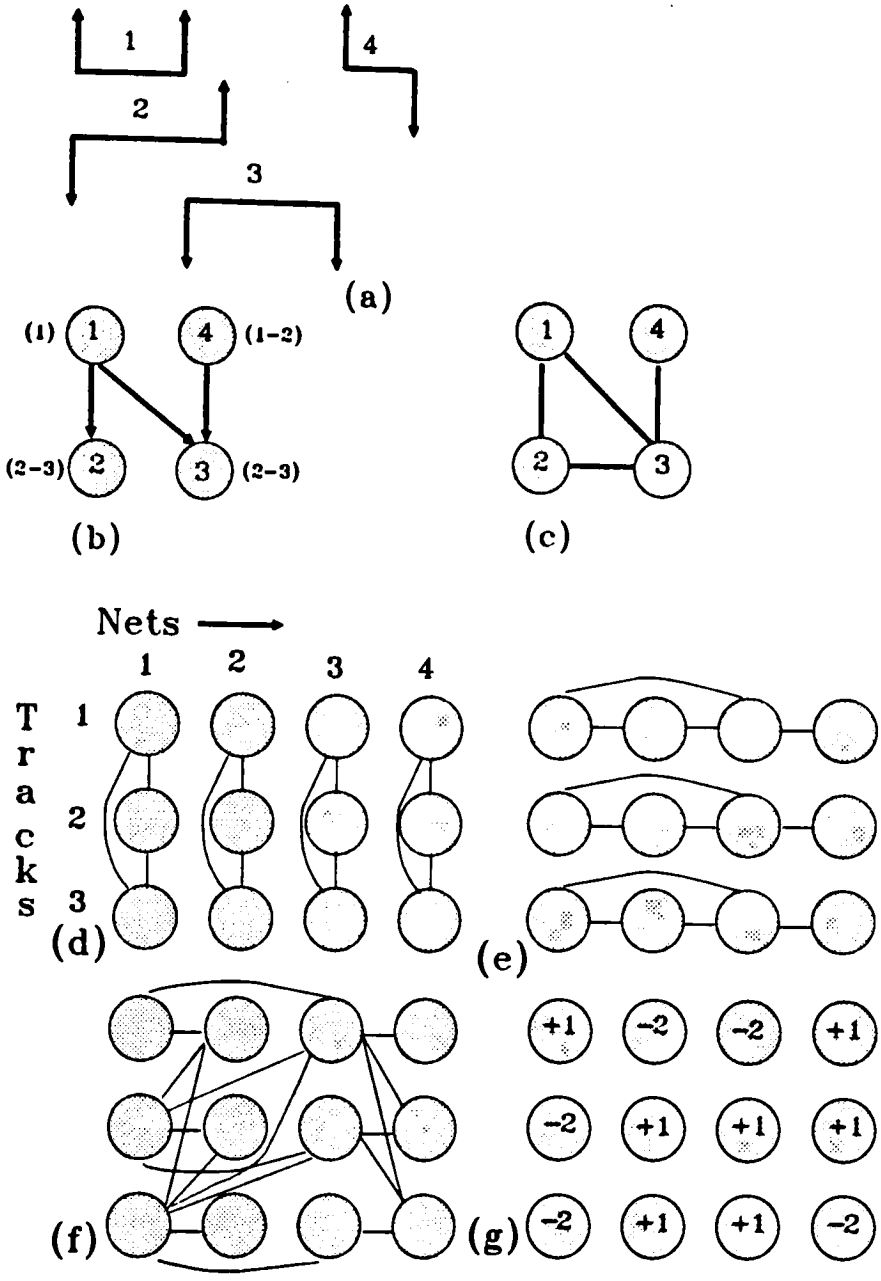


Figure 2.12: In part (a), the problem instance is shown, with the vertical constraint graph in part (b) and the horizontal constraint graph in part (c). The feasible tracks for each node are shown adjacent to it, enclosed in parenthesis, in the vertical constraint graph. We assume that a solution is possible in three tracks. Twelve neurons are, therefore, needed. Each edge (interconnection) in the diagram (parts d, e and f) is inhibitory and has a weight of -2. Part (d), shows the edges due to the constraint that a single net be assigned to not more than one track. Part (e), shows the edges due to the horizontal constraints, and, part (f) shows the edges due to the vertical constraint graph. Part (g) shows the external inputs (biases) which are determined from the feasible tracks of each node. The edges arising from the constraints are shown separately for the sake of clarity.

Problem Instance	Initial State					
	Zero			Random		
	Min	Max	Max.%	Min	Max	Max.%
ex1(21)	15	20	95	15	19	90
ex3a(45)	34	41	91	34	41	91
ex3b(47)	32	43	91	33	42	89
ex3c(54)	41	49	91	40	48	89
ex4b(57)	45	52	91	43	50	88
ex5(63)	57	61	97	53	60	95
dif(72)	50	60	83	50	62	86

Table 2.6: This table shows the minimum/maximum number of nets routed, without conflicts, by the discrete Hopfield network. The numbers in parenthesis are the number of nets in each problem instance. When the the initial state was zero for all the neurons, a single epoch was needed to reach equilibrium, provided the inhibitory weights are -2 and the excitory weights are +1. For the case of randomly initialized state, upto four epochs were needed. It should be noted that in the majority of the cases, 90% of the nets can be routed in a single epoch, by starting from the state with all neurons **off**. The experimental results obtained were over 100 trials.

```
function Hopfield_relax  returns consensus
/* Function for simulation of a Hopfield network. Some initial
state is assumed. After a few epochs, the system equilibrates
giving a locally maximum consensus.

N = number of nets;
M = number of tracks
V[i][j] = State of neuron ij. = 1 if net i is assigned to track j*/

do{
   nchanges = 0;
   consensus = 0;
   /* The following two loops comprises an Epoch */
   create_random_seq(net_seq, 1, N);/* 1...N randomly permuted*/
   for(k=1; k <= N; k++){
      i = net_seq[k];
      create_random_seq(track_seq, 1, M);
      for (j=1; j <= M; j++){
          t = track_seq[j];
          old_state = V[i][t];
          input = compute_sum_of_inputs_of_neuron(i,t);
          if (input > 0){
             V[i][t] = 1;
             consensus += input; }
          else          V[i][t] = 0;
          if (old_state != V[i][t])      nchanges++;
      }/* End for(j ... */
   }/* End for(k... */
}while (nchanges);
return consensus;
```

Figure 2.13: The discrete Hopfield network, as implemented above, is a simulation for an asynchronous sequential update model. All the neurons get a chance to be updated before any neuron gets a second chance. We process one net at a time, for the convenience of simulation. Experiments showed that arbitrary selection of a neuron gives equivalent results. The above algorithm can be made much more efficient by allowing only feasible tracks and by recognizing that no more than one track can be assigned to a neuron. Note that V[i][t] = 1 implies net i has been assigned to track j

```
function compute_sum_of_inputs_of_neuron(n, t)
/* This function assumes an adjacency list representation of the
horizontal and vertical constraint graph. For the latter we need
both a list of predecessors and successors.
HCG = Horizontal Constraint Graph
VCG = Vertical Constraint Graph
HW  = Weight for horizontal constraints
CW  = Weight for column constraints
VW  = Weight for vertical constraints
*/

     C = 0;  /* Count column constraint violation */
     for (k=1; k <= M; k++)
       if (k != t && V[n][k])
          C++;
     HC = 0; /* Number of horizontal conflicts */
     for each net k adjacent to n in the HCG
        if (V[k][t])
            HC++;
     VC= 0; /* Number of Vertical conflicts */
     for each net k a predecessor of n in the VCG
        if track occupied by k >= t
            VC++;
     for each net k a successor of n in the VCG
        if track occupied by k <= t
            VC++;

     Sum = ExtInput[n][t] + CW * C + HW * HC + VW * VC;

     return Sum;
```

Figure 2.14: The simulation algorithm computes the sum of the inputs for neuron (n,t). It can be made more efficient by recognizing that a single net cannot be assigned to more than one track and that all we need to know is whether, or not, there is a conflict with one of its neighbors.

2.6.3 Application of the Boltzmann machine to the Channel-Width Minimization Problem

The Boltzmann machine was discussed in Section 2.4. It is a generalization of the discrete Hopfield network. As remarked in the previous section, the Hopfield network follows a greedy update rule allowing an update only if it increases the consensus. In the Boltzmann machine, the updates are not deterministic but depends on the change in consensus as defined in Eq. (2.10) and a control parameter called the temperature. The probability with which a state is changed is given by a logistic function shown in Fig. 2.4 and defined by Eq. (2.11). We should remark here that the change in consenus can assume the following sequence of values: ...,+5,+3, +1, -1, -3, -5,..., etc., i.e., $\mid \Delta \mathcal{C} \mid \geq 1$. Consequently, as seen from Fig. 2.4, reducing the temperature below 0.1 will have very little effect on the acceptance probability. This is in sharp contrast to published results where the recommended stopping criteria for the temperature is of the order of 10^{-5}. A typical algorithm for a Boltzmann machine simulation is shown in Fig. 2.5. Our implementation of the Boltzmann machine, for our experiments, is shown in Fig. 2.15 and Fig. 2.16. We have split the simulation into two parts–the core and the controller. The core portion simply executes a single epoch at a given temperature and returns the number of neuron state changes. The controller part decreases the temperature and determines, whether or not, the stopping criteria has been satisfied. This strategy allows us to experiment with different stopping criteria. Experimental results with the Boltzmann machine are shown in Table 2.7. As reported in the literature, the simulation time for the Boltzmann machine is very long. Theory guarantees convergence to optimal solutions only for the asymptotic case. However, the introduction of discretization destroys the guarantee of aysmptotic convergence. For the Boltzmann machine to produce any

```
function BM_core(T)
/*  Boltzmann Machine Core. Some initial state and temperature
is assumed. This function returns after a single epoch of the
Boltzmann machine at the control parameter value T. It returns
the number of changes that occurred in a single epoch.
N = number of nets;
M = number of tracks
V[i][j] = State of neuron ij
T =       Temperature */
   nchanges = 0;
   /* The following two loops comprises an Epoch */
   create_random_seq(net_seq, 1, N);/* 1...N randomly permuted*/
   for(k=1; k <= N; k++){
      i = net_seq[k];
      create_random_seq(track_seq, 1, M);
      for (j=1; j <= M; j++){
          t = track_seq[j];
          delta_C = compute_sum_of_inputs_of_neuron(i,t);
          if (V[i][t])
             delta_C = -delta_C;
          X = -delta_C / T;
          if      (X <= -5.00)      A = 0.999;
          else if (X >= 5.00)       A = 0.0001;
          else                      A = 1/(1 + exp(X));
          r = rand() % 10000;   R = (float) r /10000.0
          if (R < A){
              V[i][t] = 1 - V[i][t];
              nchanges++;
          }
      }/* End for j ... */
   }/* End for k... */
return nchanges;
```

Figure 2.15: The core of the Boltzmann machine with a logistic probability function. This function must be driven by a control program, with the value of T changed every time. Notice that since the algorithm is not entirely greedy, constraint violations can exist for some time. Consequently, in contrast to the Hopfield network, a single net may be assigned to two or more tracks.

```
/* Controller for the Boltzmann machine
QuietLimit = If no state is updated in QuietLimit successive
               epochs  the temperature is lowered.
MaxEpochs  = maximum number of (Epochs) iterations allowed
               at a given  temperature
TMin       = Minimum Temperature.
T          = Control parameter(temperature).
Beta       = decrement factor
*/
     for(T = InitTemp; T > TMin; T *= Beta){
          no_change_in_state_count = 0;
          for(epochs = 1; epochs <= MaxEpochs; epochs++){
             if (BM_core(T) == 0){
                no_change_in_state_count++;
                if (no_change_in_state_count >= QuietLimit)
                   break;
             }
             else
                no_change_in_state_count = 0;
          }
     }
```

Figure 2.16: Simulation procedure for the controller of the Boltzmann machine. The stopping criteria we have chosen here is simple. More complicated stopping criteria were also tried, with no obvious advantages.

Problem Instance	Iterations
ex1	423
ex3a	2035
ex3b	3531
ex3c	***
ex4b	***
ex5	2720
dif	***

Table 2.7: This table shows the number of epochs required for routing some of the seven benchmark problems in four layers, with the Boltzmann machine simulated using the algorithm shown in Fig. 2.16. The asterisks show that no solution could be found after 16,000 iterations. Inspite of using a wide range of parameters, an effort to find solutions to these problems was abandoned. Solutions could be found for four of the seven problems but they occurred so infrequently that averages could not be taken. For the solutions shown above, *QuietLimit* was fixed at 4 and *MaxEpoch* was fixed at 50. *Beta* was taken as 0.995 and *TMax* was set at 0.3 and *TMin* at 0.1. The simulations were run on the IBM ES/9000-900 vector-scalar supercomputer with 2.66 Gflops peak aggregate performance, at the Cornell Theory Center.

reasonable solution, it must be greedy most of the time. However, in the previous section, we have shown that the discrete Hopfield network, for the problem instances that we are dealing with, equilibrates in 3-4 epochs. It is, therefore, reasonable to question the practical usefulness of a Boltzmann machine, since it spends most of its time in equilibrium, at a local extrema, wasting machine cycles. In the forthcoming sections, we will propose different algorithms to overcome this problem.

Relax-and-Perturb Network

In this and subsequent sections, we introduce new models and study their behaviour. Recall that the Boltzmann machine is a Hopfield network with a probabilistic update rule. Most of the time it uses a greedy update rule, particularly at lower temperatures, because as we have shown in the previous section, the Hopfield network equilibrates in

3-4 epochs. Consequently, when the "ungreedy" update is taken, the system is forced away from the equilibrium state and since the new state is unstable, the system again moves towards equilibrium, perhaps to a new minima. Therefore, the "ungreedy" updates can be regarded as perturbations on the system, even if does not take place at the equilibrium state. A perturbation always increases the energy (or equivalently lowers the consensus) and in the case of the Boltzmann machine the perturbation is a caused by an unfavorable change in the state of a single neuron. The problem with the Boltzmann machine is, therefore, that it spends too much time in the equilibrium state, waiting for a lucky roll of the dice needed to produce the perturbation that will help it escape from the local minima.

Before we describe our proposed model, we introduce the concept of *clusters*. In many problems, the set of neurons can be divided into disjoint subsets called *clusters*, such that a valid solution of the problem is one in which exactly one member of each cluster is on. This, of course, is the familiar assignment problem. In the case of channel routing, we can immediately identify the clusters by looking at Fig. 2.12. Each column of neurons is a cluster. Obviously, the members of each cluster form a complete graph with inhibitory edges. This guarantees that *at most* one neuron will be on in each cluster. Note that this interconnection allows a cluster to have no neurons in the on state.

The model we will introduce in this section, we have named a "Relax-and-Perturb" network, in which we allow the Hopfield network to completely equilibrate before introducing one or more perturbations. After the Hopfield network has equilibrated, one or more nets are not assigned to any track. Other than that, there are no conflicts in the routing at this stage. The perturbation is produced by assigning a track to one of the unrouted nets, or to each of the unrouted nets. The latter case of assigning

tracks to each of the unrouted nets is certainly easier to do from the hardware point of view, because, choosing a cluster would require extra circuitry. We, therefore, envisage our neurons to be working in two phases. In the first phase – the *relaxation* phase, each neuron works in the normal manner, i.e., it sums all its inputs, including the inputs from its cluster members, and turns **on** if the sum is greater than zero. In the second phase – the *perturbation* phase, each neuron sums the inputs from its *cluster* members only and its external input. If the resulting sum is greater than zero, the neuron turns **on**. If in the *perturbation* phase, a cluster has an **on** neuron, it will remain **on**. On the other hand, if a cluster has no neuron **on**, one neuron in the cluster will turn **on** assuming sequential updates within each cluster. A conceptual sketch of a neuron in such a network is shown in Fig. 2.17. Three adders are needed. The control signal line PERTURB indicates the phase and forces the output of the first adder to zero, when active. The *relaxation* phase requires several epochs, but, the *perturbation* phase requires only a single epoch and can, therefore, be much shorter, perhaps one-third of the *relaxation* phase.

The simulation algorithm for a discrete Hopfield network relaxation was shown in Fig. 2.13. The simulation of the perturbation algorithm is quite simple and shown in Fig. 2.18.

As another variation of the above model, we can allow the perturbation to take place at the end of an epoch instead of waiting for the system to reach equilibrium. This model is actually obtained simply by reducing the *relaxation* phase time. It is difficult in the asynchronous hardware model to determine the end of an epoch, but in the case of the synchronous hardware model driven by a clock, the end of an epoch is obvious. We have named this model the *Epoch-Perturb* machine. The algorithm to simulate this machine is shown in Fig. 2.19.

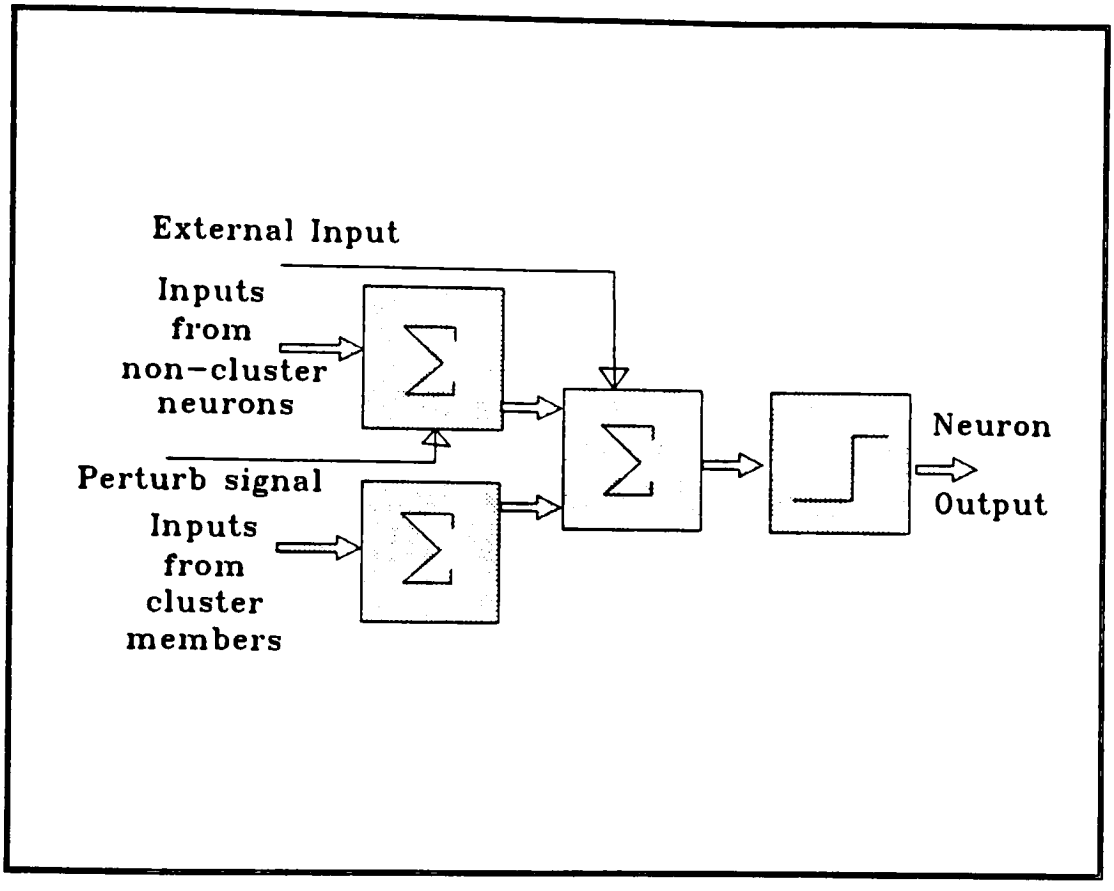


Figure 2.17: The Relax-and-Perturb Neuron Model is shown in this diagram. Notice that there are three adders and there is a control signal PERTURB which is active in the *perturbation* phase and forces the adder to output a zero. In the *relaxation* phase, all the inputs are added together to produce the internal state U_i of neuron i, whereas, in the *perturbation* phase, only the inputs from the members of the same *cluster* and the external input are added together to give the internal state. If no neuron is active within a *cluster* then one of the neurons within that *cluster* will turn on which will cause a perturbation to the system.

```
procedure  perturb( n)
{
/* attempt to assign tracks to  n nets which have not been
   assigned tracks
N = number of nets
V[i][j] = 1 if i-th net is assigned to j-th track
 */
      create_random_seq(net_seq, 1, N);
      for (i=1; i <= N; i++){
        j = net_seq[i];
        if(net j has not been assigned a track){
            r = random integer representing a feasible track
            V[j][r] = 1;
            n--;
            if (n==0)
                break;
        }
     }
}
```

Figure 2.18: The simulation procedure for the *perturbation* phase is shown. The procedure assigns one or more unrouted nets to a randomly chosen feasible track. Note that if our intention is to assign tracks to ALL unrouted nets (maximum perturbation), then we do not need the random sequence *net_seq*.

```
Function Epoch_Perturb returns the number of changes.
{
/* Do an epoch and if a cluster has no neuron, choose a neuron
randomly from within that cluster and turn it on. */

   nchanges = 0;
   create_random_seq(net_seq, 1, N);/* 1...N randomly permuted*/
   for(k=1; k <= N; k++){
      i = net_seq[k];
      assigned = 0;
      create_random_seq(track_seq, 1, M);
      for (j=1; j <= M; j++){
          t = track_seq[j];
          old_state = V[i][t];
          input = compute_sum_of_inputs_of_neuron(i,t);
          if (input > 0){
             assigned = 1;
             V[i][t] = 1;
          }
          else
             V[i][t] = 0;
          if (old_state != V[i][t])
             nchanges++;
      }/* End for(j ... */
      if (assigned == 0)
         nchanges++;
   }/* End for(k... */

   perturb(N); /* Do the perturbation of all unassigned nets*/
   return nchanges;
}
```

Figure 2.19: A simplified algorithm for the Epoch-Perturb simulation is shown above. Our implementation is a more efficient version of this algorithm. The function *perturb* is given in Fig. 2.18.

2.6.4 The MaxNeuron Machine

In this section, we propose the *MaxNeuron* machine, in which at the end of each epoch, every cluster has *exactly* one neuron **on**. The neuron that is fired within a cluster is the neuron that has the maximum value of the sum of its inputs, among the cluster members. Notice that the *maximum* neuron fires, even if the sum of its inputs is negative, and, this is precisely where the perturbation comes from. Unfortunately, because of the greedy approach, the same neuron n may be selected over and over again, although the optimal solution may require that neuron n be in the *off* state. Such an unfortunate situation was observed in many of our experiments. To overcome this problem, we randomly choose a member of a cluster if the *maximum* neuron has a negative sum of inputs and it is already **on**. Obviously, the MaxNeuron machine is more complicated than the ones proposed in the previous section. A variation of the MaxNeuron machine is the following: For each cluster c, fire one neuron which has a positive sum of imputs; if no such neuron exists, then randomly fire one of the neurons in cluster c. We have actually experimented with this model and found its performance quite satisfactory. An algorithm simulating the MaxNeuron is shown in Fig 2.20. The algorithm implicitly assumes that each cluster has exactly one neuron **on** all the time. If this condition does not hold at any time, the algorithm will fail.

```
Function MaxNeuron   returns number of changes
/* This MaxNeuron function performs one epoch */
   nchanges = 0;
   create_random_seq(net_seq, 1, N);/* 1...N randomly permuted*/
   for(k=1; k <= N; k++){
      i = net_seq[k];
      /* fire a neuron in the i-th cluster */
      p = current neuron within cluster i that is on;
      max_t = neuron in cluster i with max sum of inputs
      max_input = input of max_t
      if (p != max_t){
         V[i][p] = 0;
         V[i][max_t] = 1;
         nchanges++;
      }
      else if (max_input < 0){
        r= randomly chosen feasible track of net i;
        V[i][p] = 0;
        V[i][r] = 1;
        nchanges++;
      }
   } /* end k = 1 ... */
 return nchanges;
```

Figure 2.20: The simulation algorithm for the MaxNeuron machine. Note the the random choice of a neuron if the *maximum* neuron is already on and the sum of its input is negative. The MaxNeuron machine obviously attempts to assign a conflict-free track to a net, failing which, it assigns a track to a net such that the number of conflicts is minimum.

2.6.5 The Swap Machine

We introduce yet another model, primarily inspired by the successful 2-opt and 3-opt heuristics proposed by Lin and Kernighan [48]. This is primarily a theoretical model, because we do not, at this time, propose a method for building it. The simulation algorithm for this machine is shown in Fig 2.21. It essentially attempts to assign a track to a net with no conflicts, if possible; otherwise, if the confict is caused by a vertical constraint then the two nets exchange their tracks, if feasible. If all this fails, a feasible track is randomly chosen and assigned to a net. An enhancement to this algorithm would be replace the randomly chosen track by a track with the least number of conflicts, similar to the MaxNeuron machine.

```
Function SwapMachine   returns number of changes
/* This MaxNeuron function performs one epoch */

   nchanges = 0;
   create_random_seq(net_seq, 1, N);/* 1...N randomly permuted*/

   for(k=1; k <= N; k++){
      i = net_seq[k];
      p = currently active neuron in cluster i
      if  neuron p has sum of inputs > 0 then
         continue with the next iteration of the loop
      nchanges++;
      if a neuron t in cluster i exists such that its sum of
              inputs > 0  then
           V[i][p] = 0;     V[i][t] = 1;
      else if one of the conflicts is due to a vertical conflict
           with net j and the tracks assigned to nets i and j are
           feasible when  swapped then
             swap the tracks assigned to i and j
      else randomly assign a feasible track to net i.
   } /* end for k = ... */
```

Figure 2.21: The simulation algorithm for the Swap machine is shown above. The performance of this machine is much better than any of the previously discussed machines. At this time, this machine should be regarded as a theoretical machine only.

2.6.6 Experimental Results

We have extensively experimented with our proposed machines. The programs were written in C and developed with the help of Borland's Turbo C compiler in the MS-DOS environment. Although the PC was used for all the developmental work, many runs were made on the IBM ES/9000-900 vector-scalar supercomputer at the Cornell Theory Center, particularly with the Boltzmann machine which was quite adamant at not providing us with the solutions. In addition to running the programs, the PC environment also allows us to visualize the final routings on a VGA monitor, and if a HP LaserJet III printer (or any other printer compatible with PCL-5) is available, the routings can be printed, some of which are shown in the figures that follow.

Table 2.8 shows the performance of our machines on the seven benchmark problems for the four-layer (two channels) routing case. Funabiki and Takefuji's results are also shown. Our implementation of Funabiki and Takefuji's algorithm did not yield a solution except for *ex1*. It should be noted that Funabiki and Takefuji could not obtain the optimal solution for the Deutsch difficult problem. We should remark here again that our convergence rates depend on the partition of the nets into two channels. The results shown in the Table 2.8 are not the best we can get but are typical. It could also get worse, if the partitioning of the nets is not favorable. The only uncertain feature in our experiments is the initial random number seed and the random number generator itself. However, we could always get optimal results within a few choice of seeds, if not on the first trial. Some four-layer routings are shown in Fig. 2.22 and Fig. 2.23 and an eight-layer routings is shown in Fig. 2.24. In Fig. 2.25a, we show the histogram for the number of iterations required to find a solution over 100 trials for the four-layer Deutsch's difficult problem by the Swap machine. It shows the number of times a solution was found with a certain range of

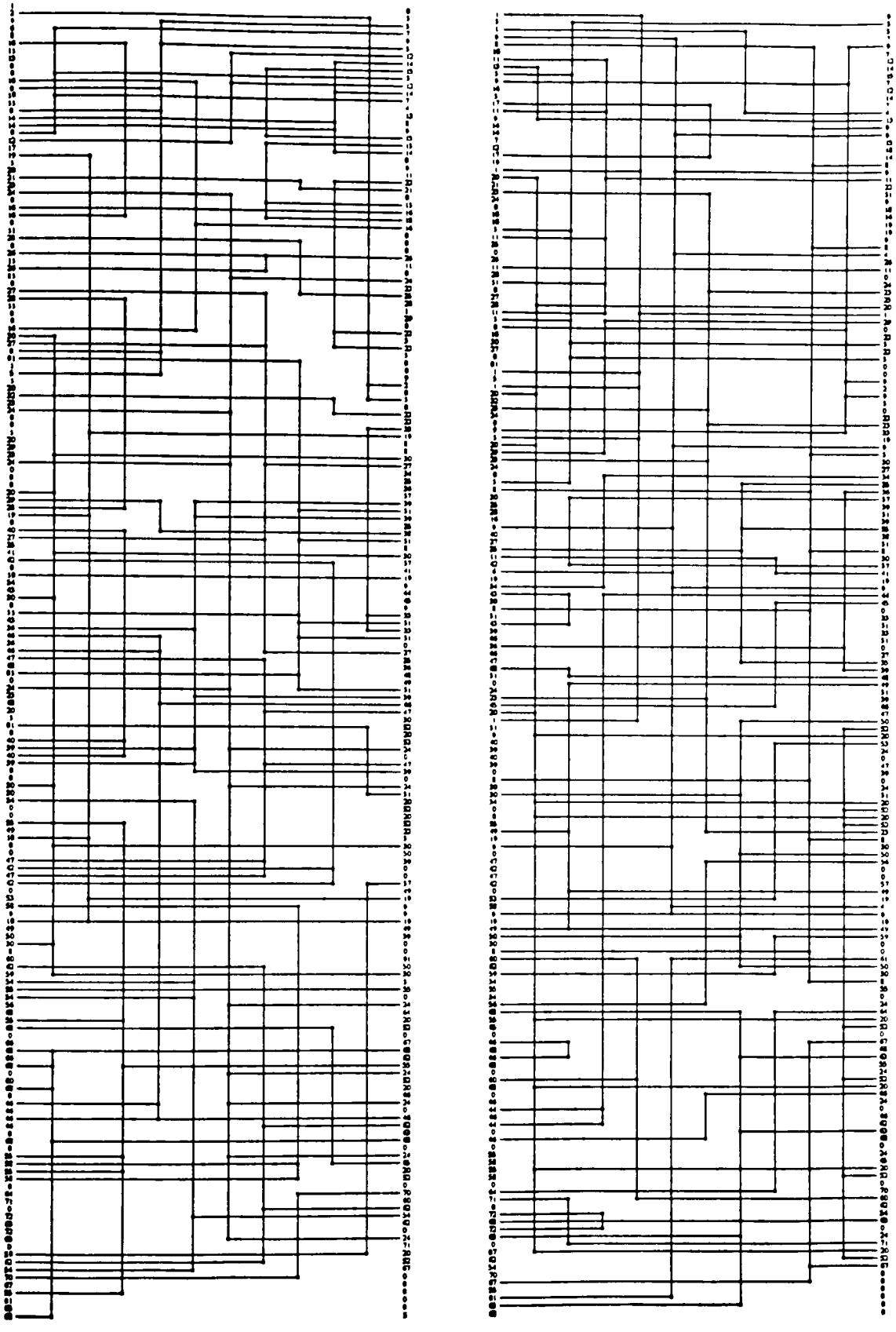


Figure 2.22: Optimal four-layer (two channel) routings as obtained by our machines for Deutsch's difficult problem. Previous results by Funabiki and Takefuji were sub-optimal.

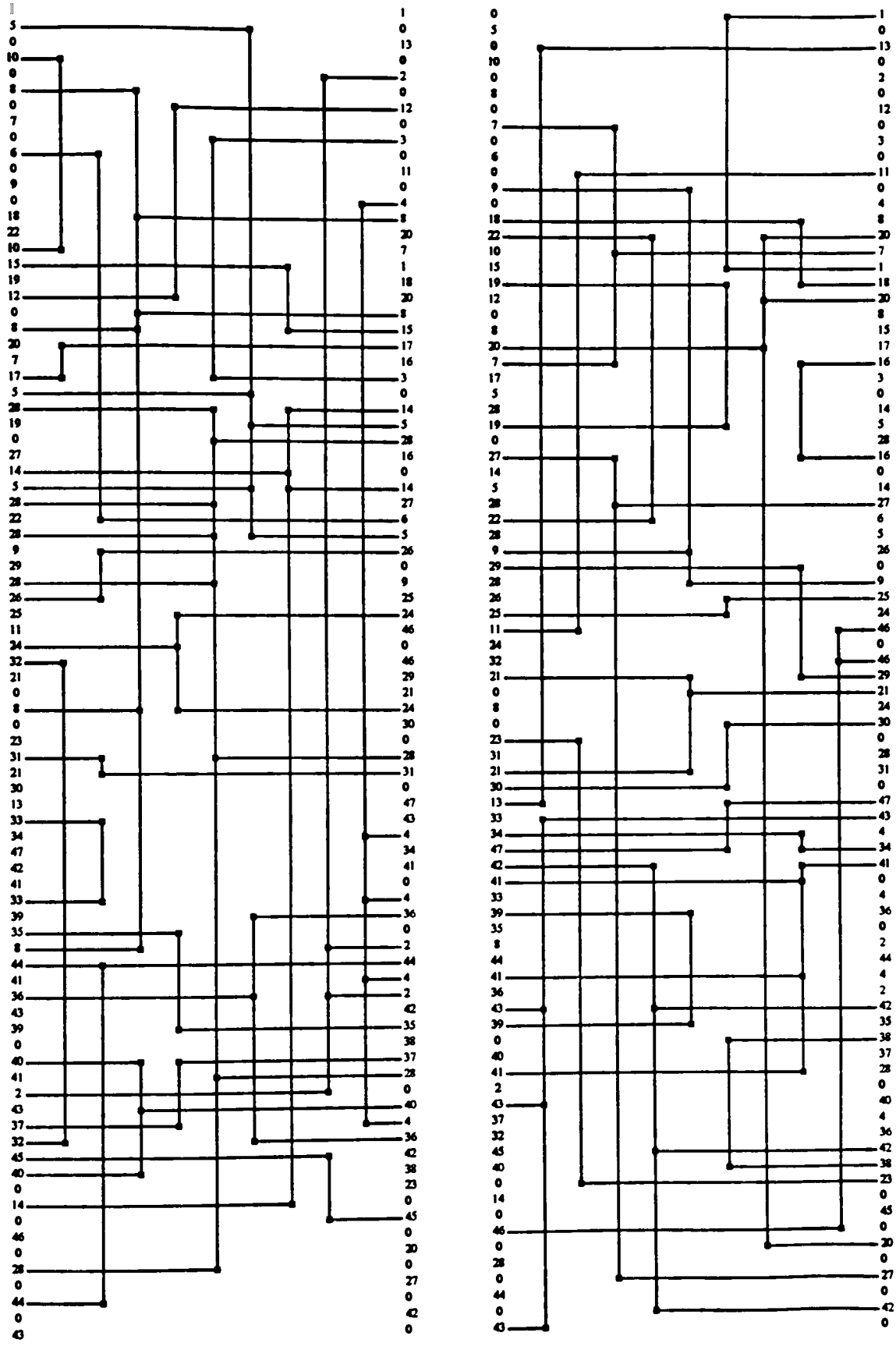


Figure 2.23: Optimal four-layer (two channel) routings as obtained by our machines for *ex3b*.

Problem	Our Machines										Funabiki and Takefuji	
	Relax Perturb 1		Relax Perturb All		Epoch Perturb		MaxNeuron		Swap			
	Avg.	Rate	Avg.	Rate	Avg.	Rate	Avg.	Rate	Avg.	Rate	Avg.	Rate
ex1	83	100	56	100	32	100	11	100	25	100	84	44
ex3a	337	100	269	100	131	100	73	100	48	100	148	29
ex3b	267	100	165	100	75	100	30	100	21	100	90	53
ex3c	608	100	395	100	278	100	108	100	63	100	280	7
ex4b	437	100	346	100	498	100	103	100	50	100	150	40
ex5	116	100	41	100	65	100	18	100	12	100	101	76
dif	2446	100	1179	100	3849	100	814	100	156	100	***	***

Table 2.8: The above table shows experimental results with our proposed machines for the four-layer (two channel) case. The results shown here are typical and varied from run to run depending on the initial random number seed. The table clearly shows the better performance of our machines. All problems were solved using an optimal number of tracks. The optimal solution of the *difficult* problem in 10 tracks is obtained for the first time. *Avg.* is the average number of iterations over 50 trials and *rate* is the rate of success, which is 100%. Funabiki and Takefuji's solution for the Deutsch's difficult problem was with 11 tracks.

iterations. From the examination of this figure, it is evident that most of the time a solution was found in less than 150 iterations. A similar figure is shown in Fig. 2.25b for the *ex5* problem in two layers (one channel). This time we compare the Swap machine with the MaxNeuron machine and it is evident that the performance of the Swap machine is better but the MaxNeuron machine is quite acceptable because it was able to find solutions that have not been found before.

In Table 2.9, we show the number of iterations required by our machines in finding a two-layer solution to some of the benchmark problems. We should remark here that finding a solution to the two- layer (one channel) problem is considerably harder than the the routing in four layers, because in the latter case, the vertical constraint graph is broken up leading to fewer constraints. To our great surprise, we were able to find very easily, the 28-track optimal solution for the Deutsch's difficult problem. Previous

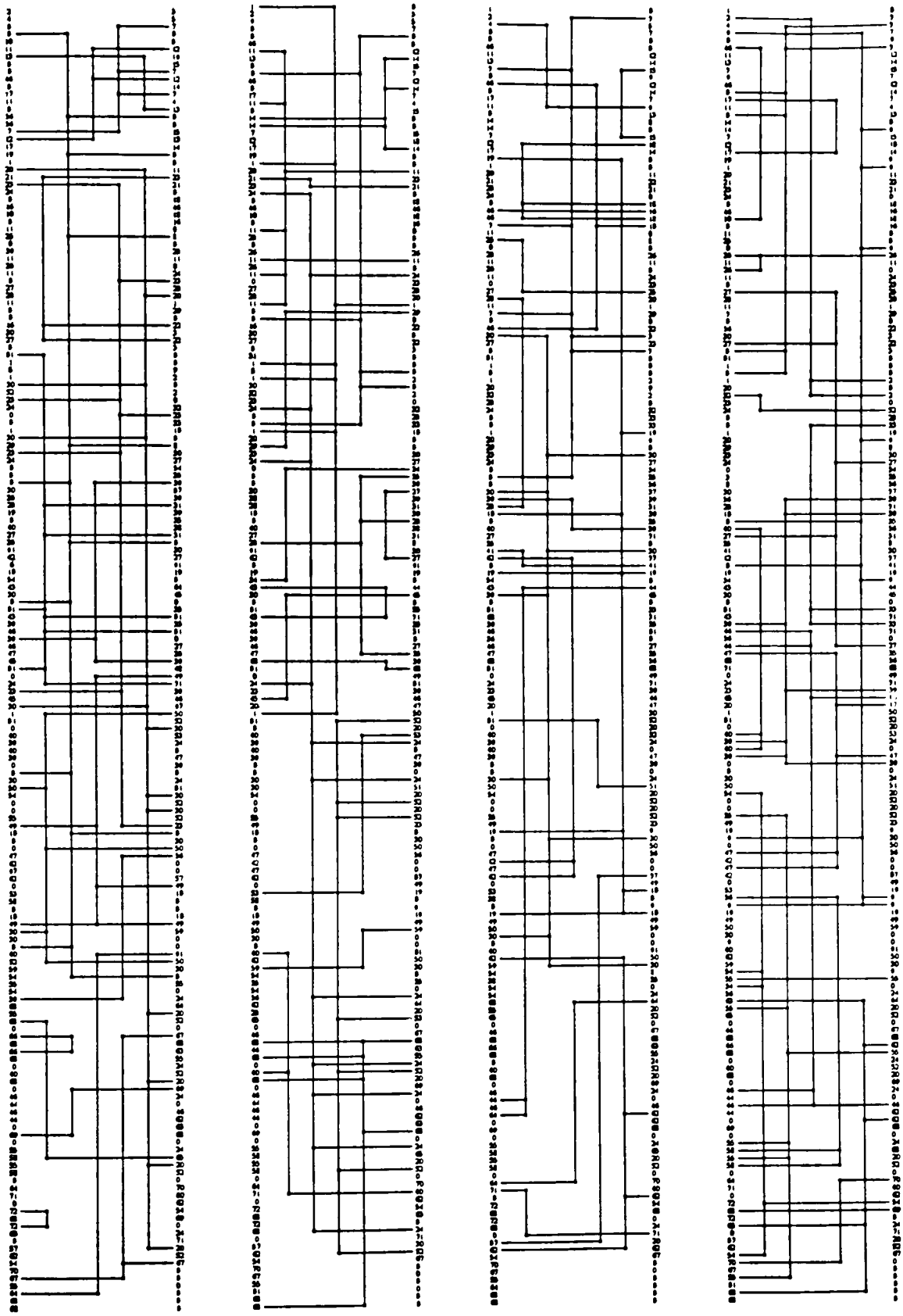


Figure 2.24: Optimal eight-layer (four channel) routings as obtained by our machines for Deutsch's difficult problem.

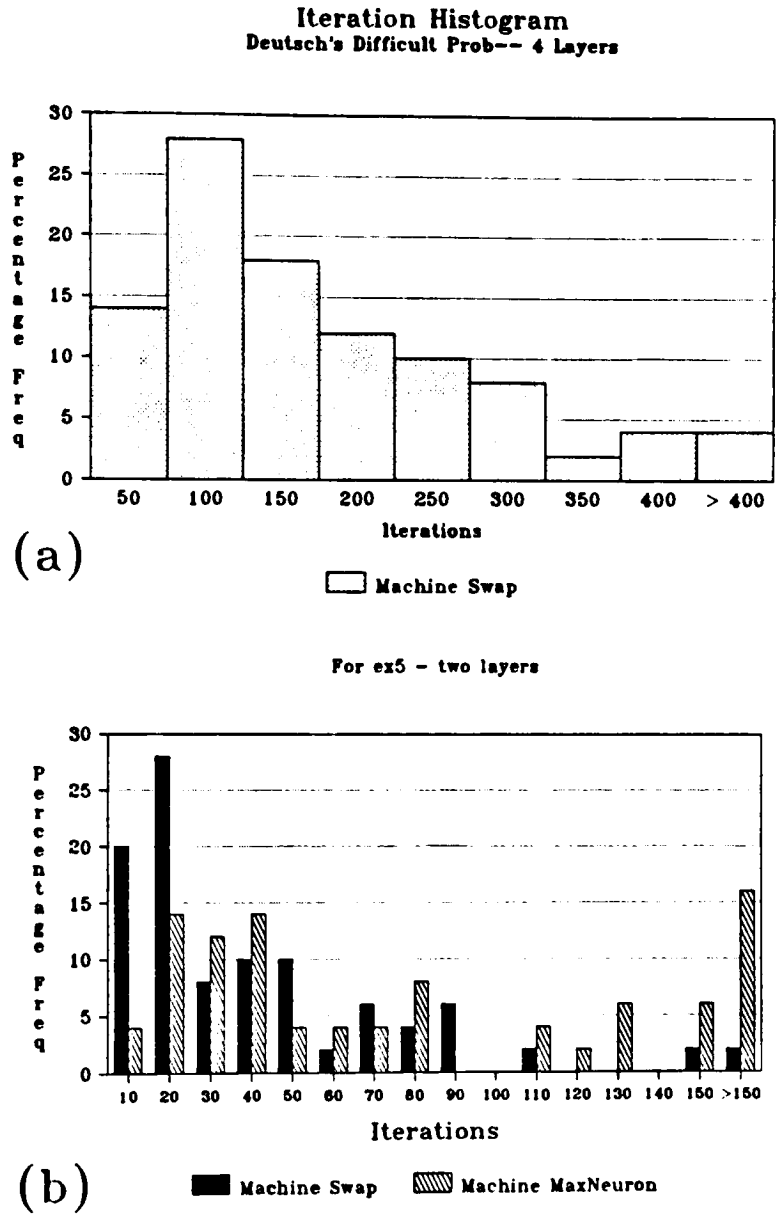


Figure 2.25: The above figure shows the percentage of solutions with a given range of iterations. In part (a), we show the percentage of trials in which a solution was found within a range of iterations for the four-layer Deutch's difficult problem by the Swap machine. In part (b), we show the same information for the problem instance *ex5* and the two machines Swap and MaxNeuron. Part (b) shows that a majority of the solutions were obtained with fewer than 30 iterations by the Swap machine. The sample size used was 100. The relative performance of the two machines are also readily apparent.

Problem	Relax-Perturb All		MaxNeuron		Swap	
	Avg.	Rate	Avg.	Rate	Avg.	Rate
ex1	99	100	24	100	24	100
ex3c	5421	25	4312	20	3687	80
ex5	153	100	100	100	45	100
dif	9900	20	8903	40	4974	65

Table 2.9: The Performance of some of our machines are shown above for the two-layer case. All problems were solved optimally. Funabiki and Takefuji did not attempt a two-layer solution since it is much harder than the four-layer solution. The last three of these problems have never before been solved with a neural network.

neural networks applied to the channel routing problem were unable to come close to solving a problem of such revered difficulty. Some two-layer routings are shown in Fig. 2.26 and Fig. 2.27.

In Table 2.10, we show the number of iterations required by our machines in finding a two-layer solution to all the problems previously solved by the network

Problem	Shih and Feng	Swap	Epoch-Relax	Takefuji
Feng1	348	1	2	10
Feng2	372	1	2	12
Feng3	164	2	1	13
Feng4	180	1	2	8
Feng5	288	2	8	34
Feng6	270	3	7	28
Feng7	284	1	6	13
Feng8	364	3	4	22
Feng9	256	2	4	16
Feng10	380	2	15	9
Feng11	448	20	32	53

Table 2.10: The table above shows the data used to demonstrate the neural network proposed by Shih and Feng. The table shows the average number of iterations over 100 trials. Our implementation of Shih and Feng's algorithm did not yield feasible solutions. The results from our implementation of Funabiki and Takefuji's algorithm are shown in the rightmost column.

proposed by Shih and Feng. The table clearly shows the superior performance of our

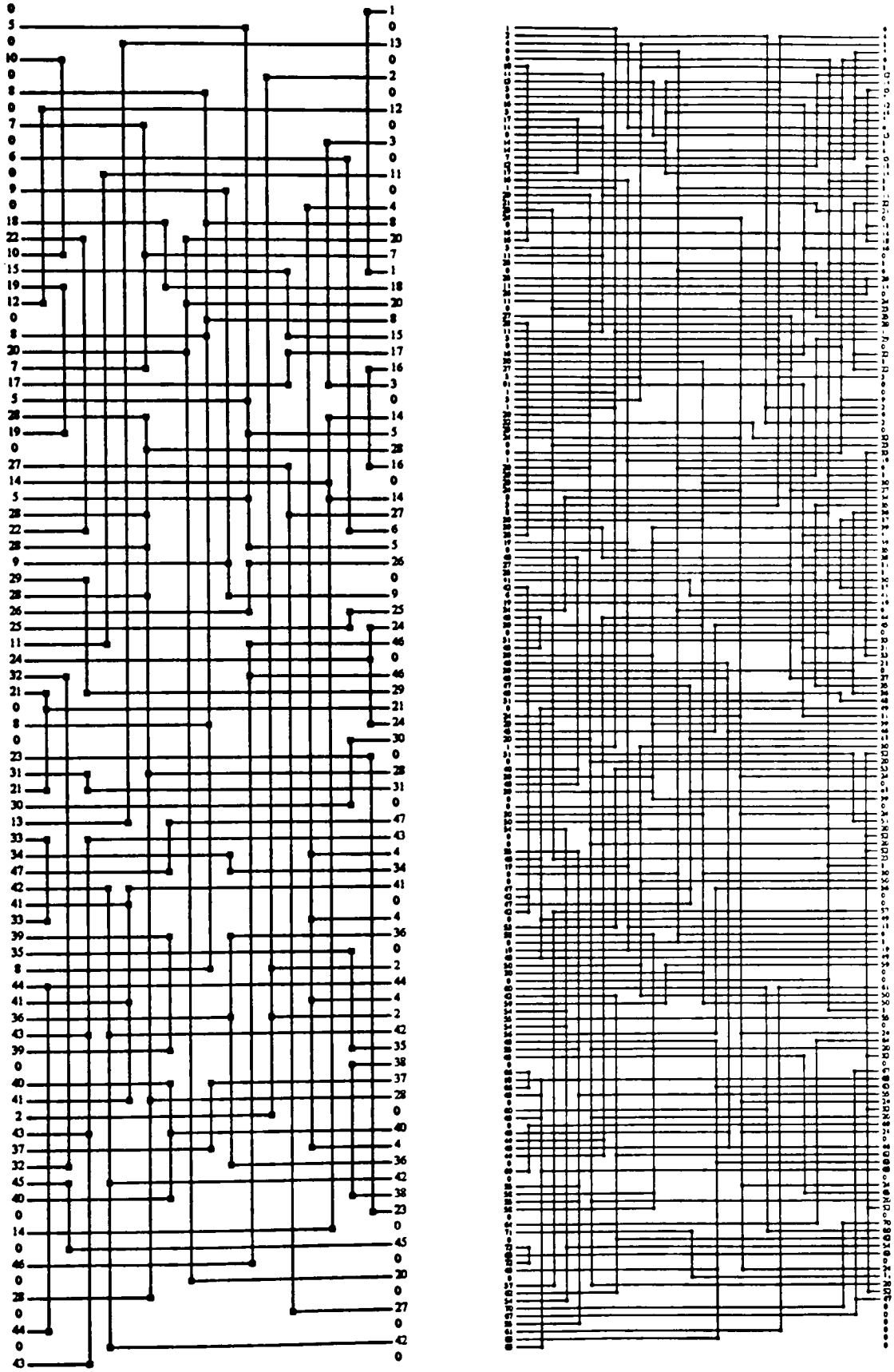


Figure 2.26: Optimal two-layer routings as obtained by our machines for Deutsch's Difficult problem (right) and *ex3b* (left).

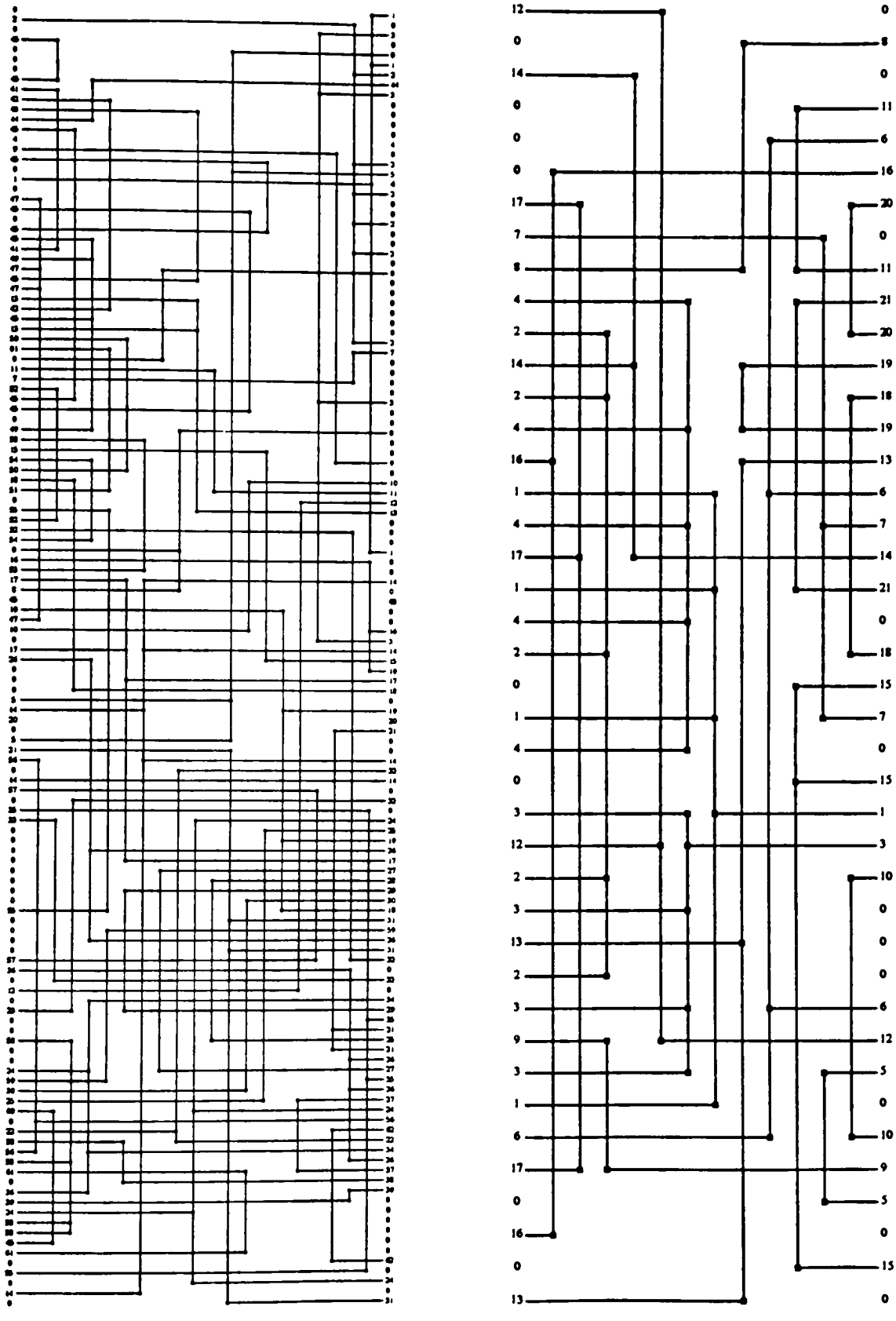


Figure 2.27: Optimal two-layer routings as obtained by our machines for *ex5* (left) and *ex1* (right).

machines. We should also remark here that when we implemented Shih and Feng's algorithm, it repeatedly converged to an infeasible solution giving conflicts in the track assignments.

In summary, we should point that

- our machines were the first to solve optimally several of the benchmark problems in two layers for the first time, including Deutsch's difficult problem.
- our machines were the first to obtain the optimal 10-track four layer solution of Deutsch's difficult problem.
- the success rates of our machines are much superior than published results.
- for multilayer routings, our machines require fewer neurons and drastically fewer interconnections.
- our machines use only two weights: +1 and -2. This makes them easier to implement in silicon.
- the simulation algorithm of our machine executes very fast.
- some of our models are practical and can be directly implemented in silicon.

Chapter 3

Via Minimization Using a Neural Network

Most present day routers use the one-direction-per-layer paradigm because they place all the horizontal segments on one layer and all the vertical segments in another layer, which results in a lot of vias (interconnection between layers). A very simple routing, that has eight vias, is shown in Fig. 3.1. Vias cause electrical instability and unpredictable timing delays. Therefore, minimizing vias, which is known to be NP-complete [40],[41], improves the reliability and performance of a chip. By allowing both horizontal and vertical segments on each layer, the number of vias can be reduced. One possible way of doing this would be to first go through the usual track assignment for each horizontal segment of the net and then when that is done, move a horizontal segment to the other layer if it does not cause an electrical short with a vertical segment. Each successful move would reduce the number of vias by two or more. Of course, the vertical segments can also be moved. The idea is to assign a layer to each horizontal or vertical segment such that the total number of

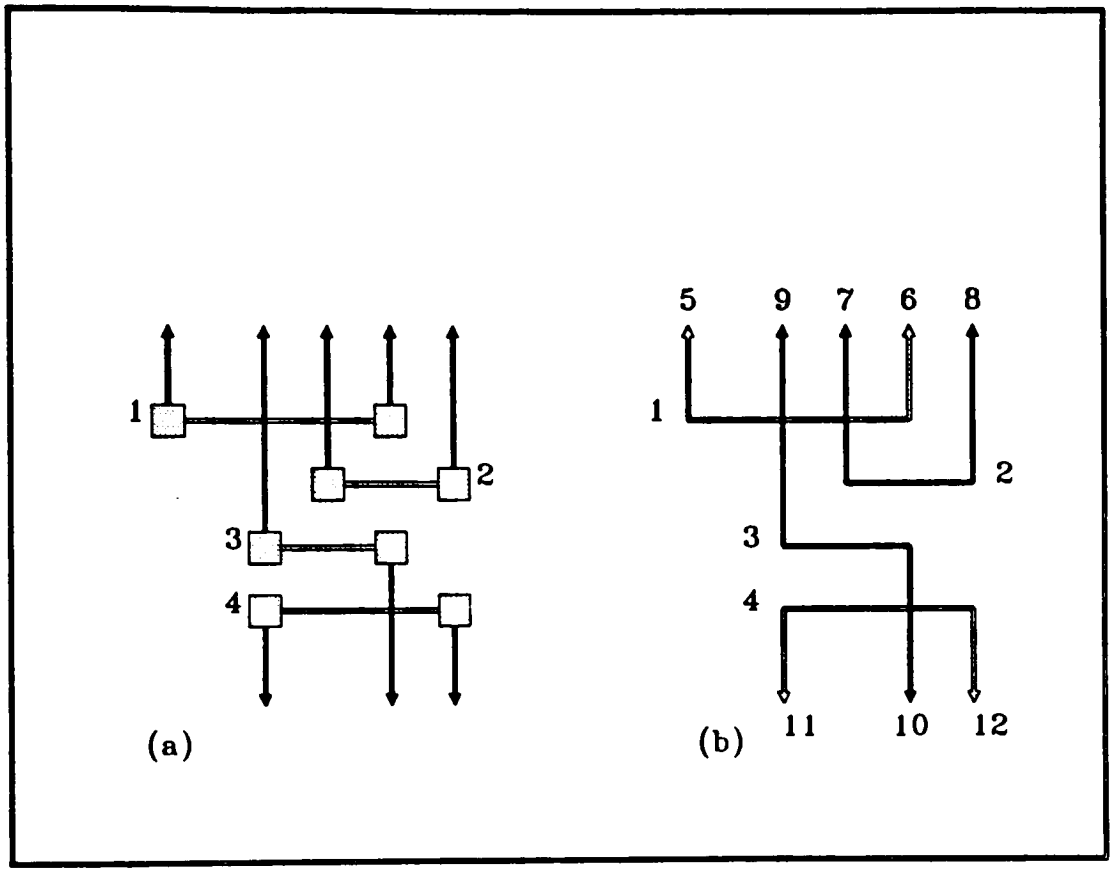


Figure 3.1: The simple channel routing example shown above is used for illustration of via minimization. There are eight vias shown as shaded squares. All the vias can be removed by properly assigning layers to each segment. In part (a), we show a valid channel routing with the horizontal segments in one layer and the vertical segments in another layer. In part (b), we show a different layer assignment for the segments. We also number the segments for later reference. For convenience, a horizontal segment is numbered the same as the net to which it belongs and segment numbers for the vertical segments are shown at the arrow heads. Notice that no vias are needed and the channel routing is no longer Manhattan. Terminals are assumed to be available in all the layers.

vias is minimized. Such a method is called *constrained* via minimization [41], [42], [43]. A second approach, the *unconstrained* or *topological* via minimization [44], [45], [46] has also been used. In this latter case, the track assignments of the horizontal segments are not done first, but instead via minimization is regarded as an integral part of the track assignment process. Better via minimizations have been obtained using this technique. As far as we know, via minimization using neural networks has not been reported before except in a Korean journal [47]. For our investigations, we used the discrete Hopfield network. It would be highly desirable to design a neural network to perform unconstrained via minimizations as well, and then compare the results with that obtained from the constrained via minimization version.

3.1 A Simple Neural Network Model for Via Minimization

Consider the example channel routing shown in Fig. 3.1, where the vias are shown as shaded squares. Each net consists of exactly one horizontal segment and two vertical segments. In conventional channel routing all the vertical segments are placed in one layer and the horizontal segments are placed in another layer. Wherever a vertical segment needs to be connected to a horizontal segment, a via (or a cut) is used. In addition to taking up extra space, vias are also known to cause electrical instabilities, as mentioned earlier. Consequently, it is important to minimize vias in high-performance circuits. We will approach the solution to the problem by assuming that no new vias will be introduced and, therefore, vias that are removed are the already existing ones. Also, note that all our nets are two-point nets and the algorithm that follows is for two-point nets only, but if applied to multipoint nets, it will give a

solution, which, however, may never be optimal. These restrictions will be removed later on to give a very general model. The solution to this simple case is based on the idea, that a horizontal or a vertical segment may switch layer as long as it does not intersect with a segment of a different net because that would produce an electrical short. By switching layers, segments of the same net may end up in the same layer, with a concomitant reduction in the number of vias. It is not difficult to see how a neural network can be utilized for the layer assignment of each segment. A neuron will be used to represent each segment and the neuron state will indicate the layer assignment of the corresponding segment. Two neurons will be connected by a negative (inhibitory) weight if the corresponding segments instersect and belong to two different nets. In other words, both of these neurons will be discouraged from assuming the same state since that will cause a short circuit. A neuron representing a horizontal segment h belonging to the net n, will be connected by a positive (excitory) weight to other neurons that represent vertical segments belonging to the same net n. Therefore, two neurons that represent segments connected by a via will tend to be assigned to the same layer thus removing the via. The positive weight may be taken as $+1$, whereas, a negative weight should be larger in magnitude than the maximum number of possible vias in any single net to ensure that a single short circuit will determine the sign of the sum of the inputs to a neuron. These ideas are illustrated in Fig. 3.2.

The neuron output is assumed to have bipolar values of $+1$ in the on state and -1 in the **off** state. The input U_i to neuron i, is the weighted sum of the outputs V_i of the adjacent neurons, i.e. $U_i = \sum_{j \epsilon \text{adj}(i)} W_{ij} V_j$. The output V_i is related to the

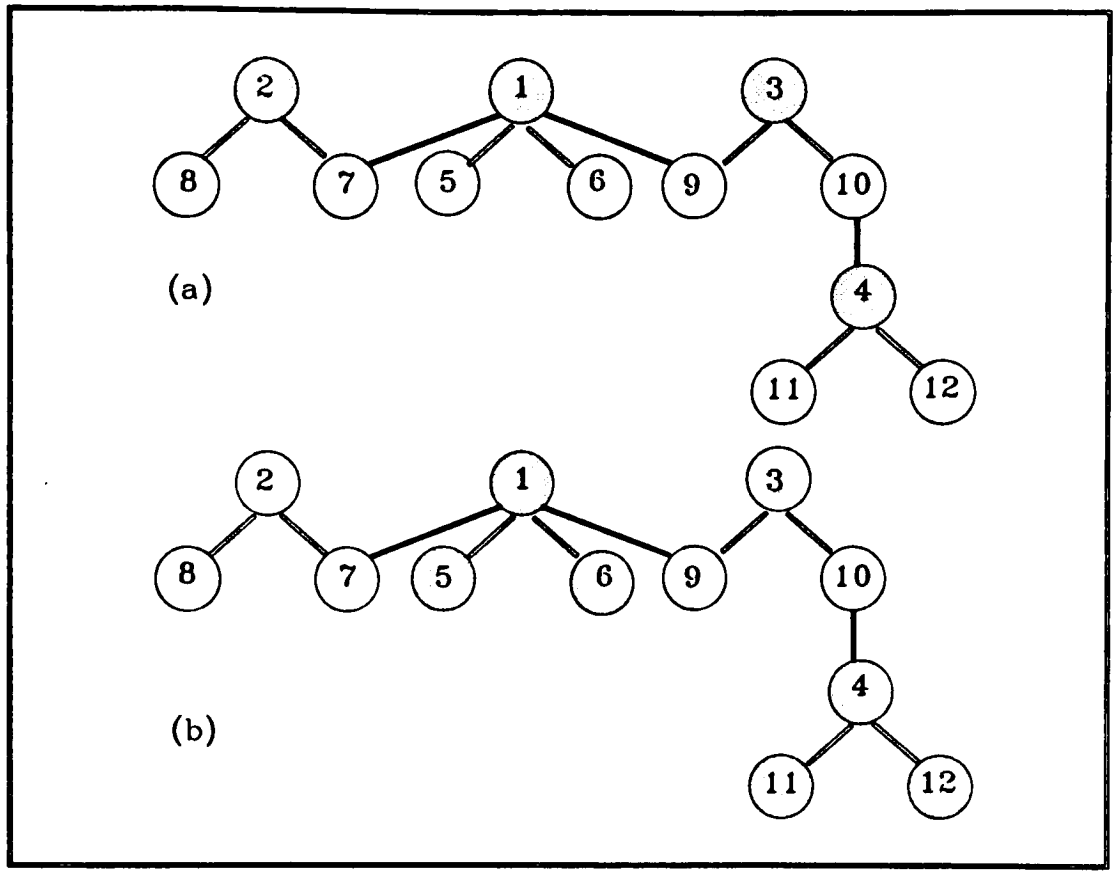


Figure 3.2: The neural network representation of Fig. 3.1. The shaded neurons are in the -1 state and the unshaded neurons are in the +1 state. The initial horizontal layer is -1 and the initial vertical layer is +1. The dark edges are inhibitory connections and the light edges represent the excitory connections. If two neurons connected by a dark edge ends up in the same state, then there is a short circuit. If two neurons connected by a light edge ends up in the same state, then a via has been eliminated. Part (a) shows the initial configuration of the neurons corresponding to the layer assignment of Fig. 3.1. For each neuron, we then sum its input and assign the *sign* of the sum to the neuron state. When no more changes take place, the system is at a local minima. Part (b) shows the final state for this example. It shows that no vias are required.

input by

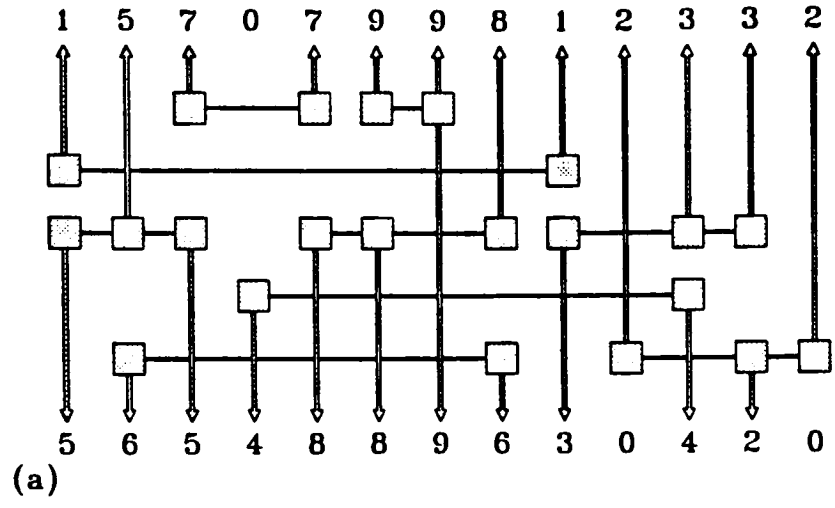
$$V_i = \begin{cases} 1 & \text{if } U_i > 0 \\ -1 & \text{otherwise} \end{cases} \tag{3.1}$$

A feasible solution is one which does NOT have short circuits. Out of all the feasible solutions, we would like to find one which has the smallest number of vias. As is well known, the Hopfield network converges to a local minima and not a global minima. In order to overcome this problem, the Hopfield network is allowed to reach equilibrium, and the consensus, which is the sum $\sum_{i=1}^{N} U_i V_i$, is noted. The network is then perturbed slightly by changing the states of a fraction of the neurons and the relaxation process repeated 100-200 times. For small instances, the via reduction is dramatic, although the algorithm designed was just for two-point nets. Table 3.1 shows the results of our experiments for this simple case. Fig. 3.3 shows that it does

Problem Instance	Number of Nets	Number of Vias	
		Initial	Final
feng5	9	22	5
feng6	10	22	3
feng7	10	22	2
feng8	10	19	6
feng9	10	23	3
feng10	10	27	5

Table 3.1: The two-point net via minimization algorithm applied to problem instances from [36]. They are all optimal except for *feng5*.

not give optimal results for a layout with multipoint nets. The via at the left end of net *8* can be removed. The algorithm was extended to handle multipoint nets, which yielded optimal results for all the instances given in Table 3.1



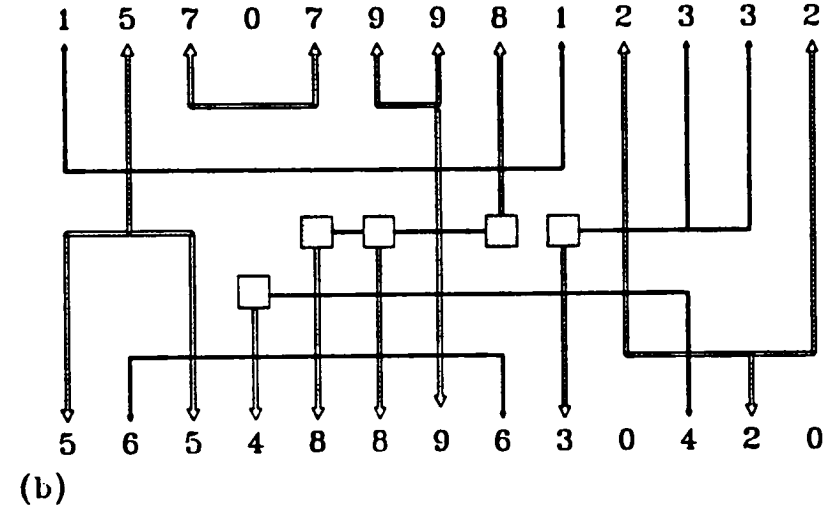


Figure 3.3: In part (a), we show the original layout of *Feng5* with 22 vias, and, in part (b), we show the results of our via-minimization algorithm for two-point nets applied to three-point nets. Note that the minimization is not optimal, because the leftmost via on net *8* can be removed. The hollow lines are in one layer and the dark lines are in another layer.

3.2 A Cluster Graph Approach to Via Minimization

So far we have used a very simple approach to via-minimization. In this section, we introduce a more complicated model for via-minimization. To start with, we should recognize that if a segment s does not intersect with a segment of another net then we should be able to move segment s to the other layer. Thus, such a segment s can be combined with another segment s' that is via-connected with s, consequently removing a via and decreasing the number of segments. For example, in Fig. 3.1, since segment *5* has no intersection, it is moved to the same layer as segment *1*, thereby removing the via between them. It is immediately seen that each of the nets *1,2* and *4* will end up with all its segments on a single layer. Net *3* is a little different. We can combine segment *3* with segment *9* and but have to leave a via between segments *3* and *10*, which will actually be removed upon further consideration. At this point, we have four "super" segments {*1,5,6*}, {*2,7,8*}, {*3,9*}, {*4,11,12*} and a single "primitive" segment {*10*}. We now have to assign a layer (or color) to each one of these modified segments. This is most easily done by building an intersection graph which helps avoid short circuits when the coloring is done. These concepts are shown in Fig. 3.4. The method just described is not very flexible, because, at times, we have to introduce vias where none existed before. Therefore, it is convenient to start with a Manhattan solution but with no layer assignment as shown in Fig. 3.5a. We then introduce can*didate* vias by using a very simple rule: Place a via between every two intersections (crossings), if space permits. This allows the intersecting segments to change layer which may help in minimizing the number of vias globally. Most of the segments can be immediately numbered simply by considering the lines that are

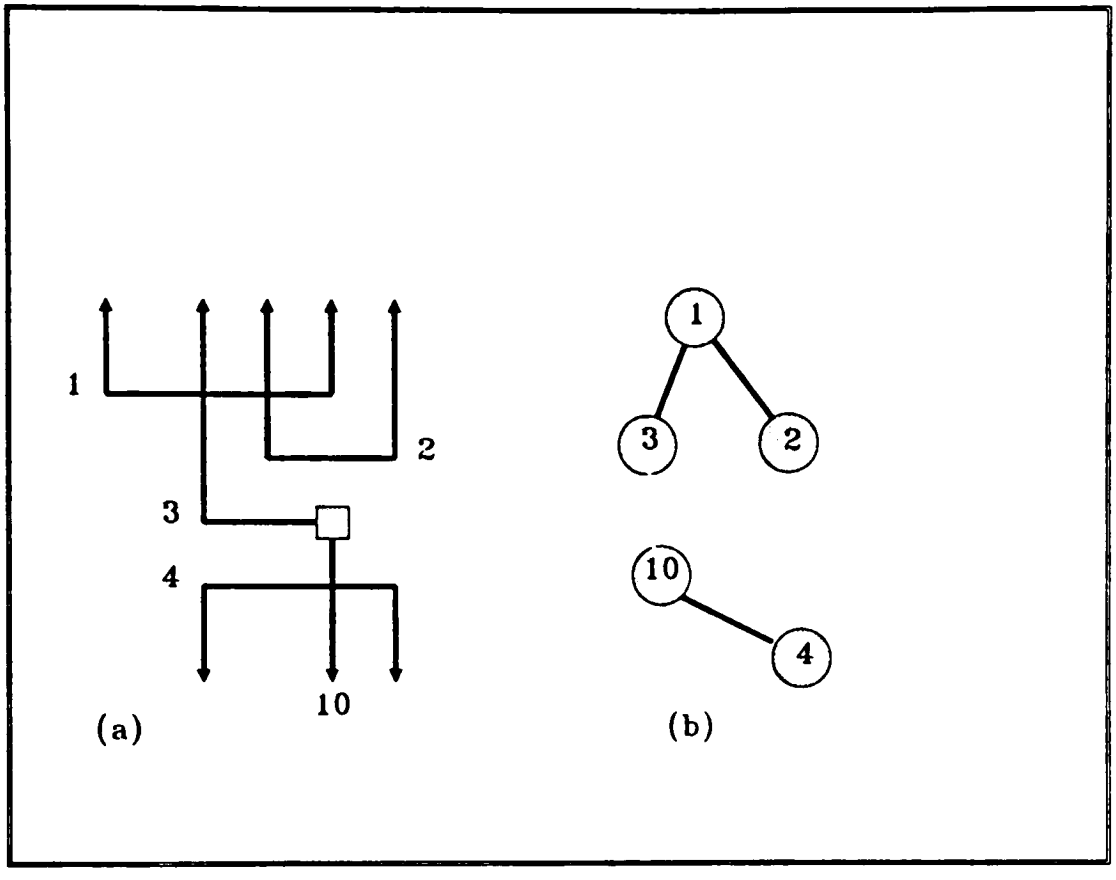


Figure 3.4: In part (a), we show the problem instance given in Fig. 3.1, with each segment not assigned to any particular layer. The non-intersecting segments have been combined with an intersecting segment leaving only five concatenated segments to color. In part (b), we show the intersection graph. The dark edges correspond to intersections and the light edges correspond to via connections. We have to color these nodes such that there are no short circuits and the number of vias is minimum. The short circuits can be easily avoided by coloring a node with some color and coloring adjacent (with respect to the intersection (dark) edges) nodes with the other color. The nodes connected by the via edges (light) are colored so as to minimize the number of vias. Carrying out this procedure for the above case yields zero vias.

incident on a via. Each segment can be vertical, horizontal, L-shaped, U-shaped, etc. A segment in the present context is, therefore, a maximal piece of wire which either does not need a via or cannot contain a via. Notice the two vias *I* and *F* in Fig. 3.5b. They were placed there by our rule and will ultimately get removed. They are, however, not needed, because net *1* can move to either layer, and, so is the case with net *11*. Therefore, the segments involved with these vias are not all numbered since they will not be discussed further. As shown by via *J*, pairs of intersections can share the same via. The only requirement is that between every two intersections there be a via. For example, via *J* is the via between instersections 15,22 and 11,16, and, it is also the via between intersections 24,17 and 11,16. Segment *18* is different from all the other segments in Fig. 3.5b, because it contains no intersection. If segment *18* is combined with some other segment while building the intersection graph, one extra via may be introduced. To understand the reason for this, suppose the segments incident on via *J* are all colored red and the segments incident on via *K* are all colored blue, then there is no via. But since segment *18* is either blue or red, we will need one of the vias *J* or *K*. If we keep segment *18* as a separate node in the intersection graph, all the segments incident on the vias *I* and *J* may be coerced to assume the same color, thereby removing both vias.

3.2.1 The Intersection Graph

We now proceed to build the intersection graph as shown in Fig. 3.6a. Each node in this graph corresponds to a segment of the previous diagram. Two nodes are connected by an intersection edge (shown as a solid line), if the corresponding segments intersect and they are connected by a via edge (shown as a broken line) if the corresponding segments have a via between them. If we color node *1* RED, we

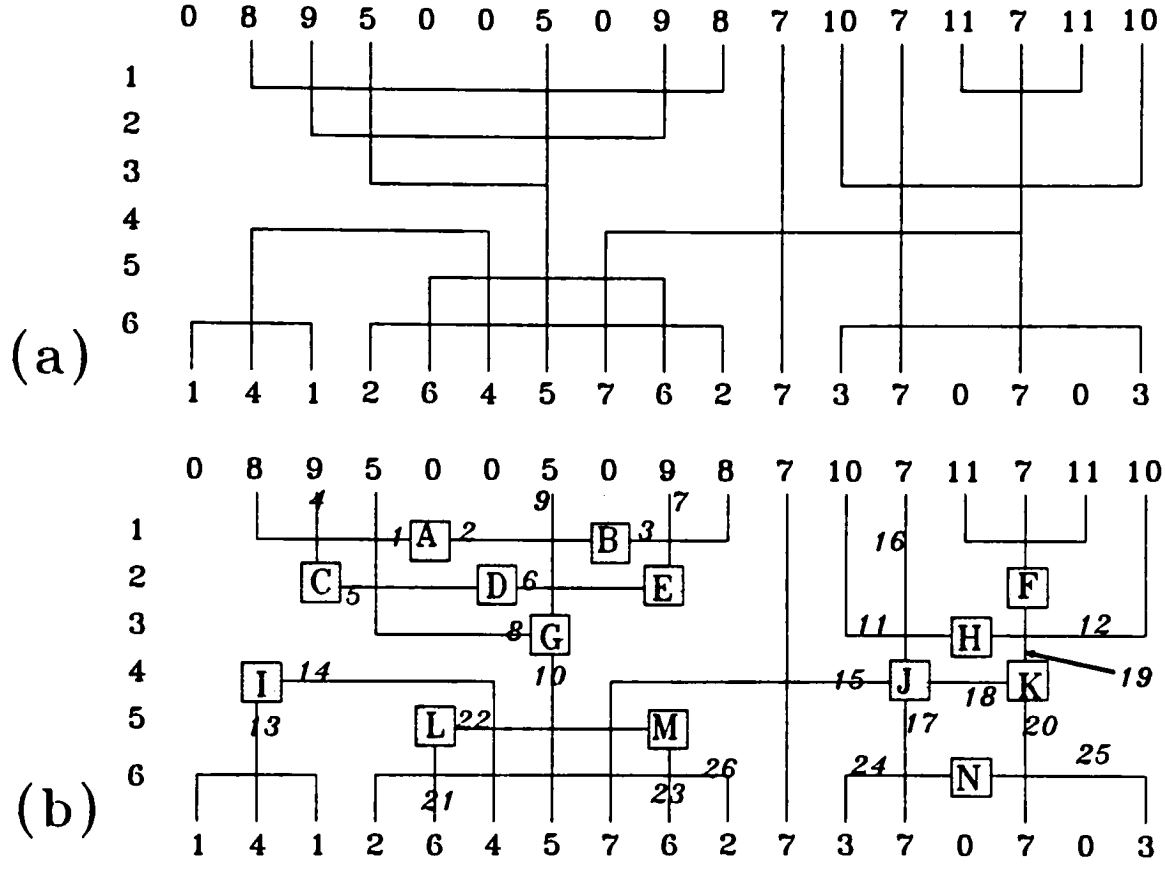


Figure 3.5: Part (a) of the above figure shows the Manhattan routing but without the layer assignment. Candidate vias are now introduced between every pair of intersections as shown in part (b). Segments are no longer straight lines. All the segments except segment *18* are intersecting segments. Segment *18* connects two vias. Nets *1* and *11* need not be considered because they can switch to either layer, as needed. Consequently, vias *I* and *F* are not needed.

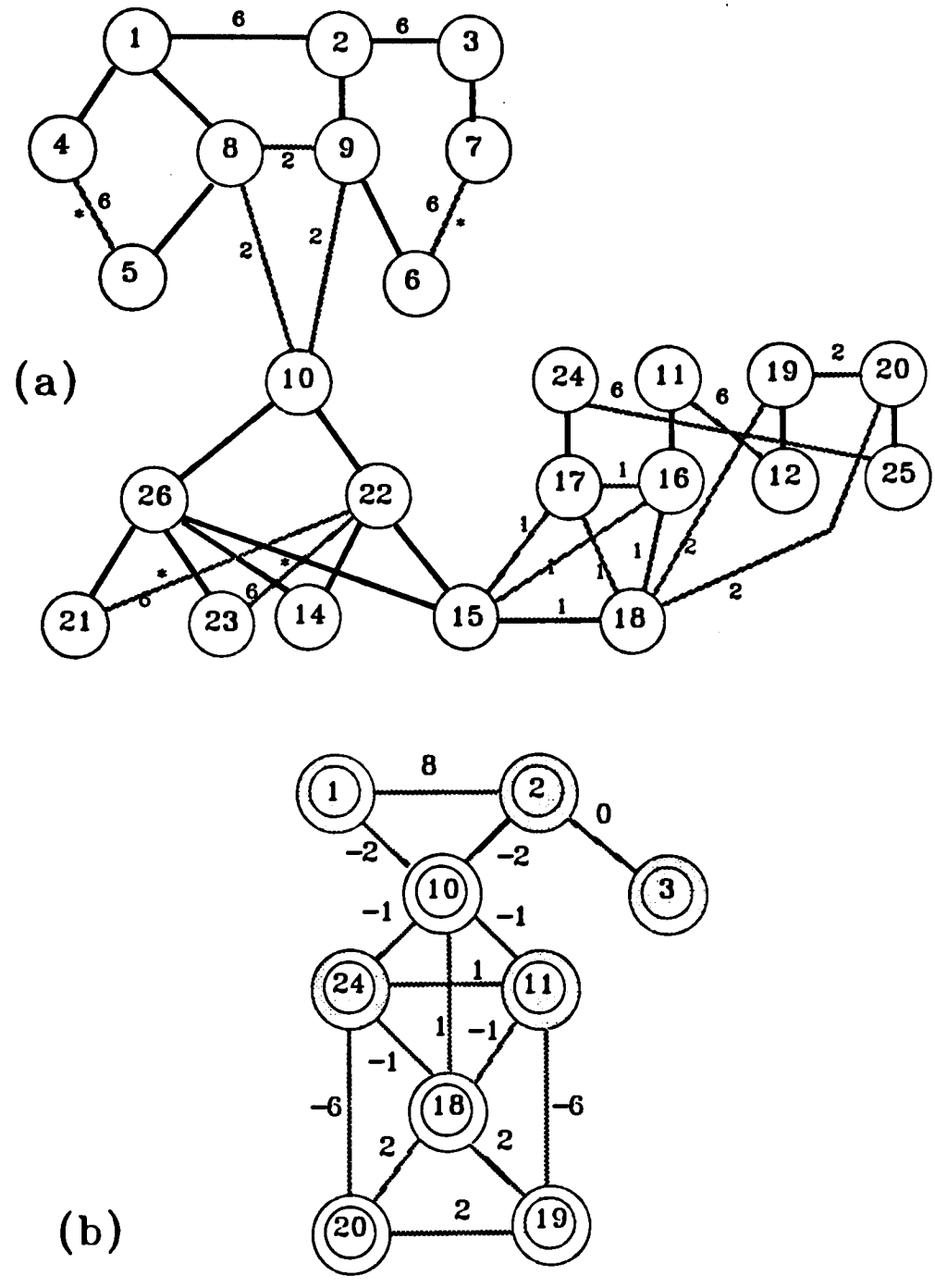


Figure 3.6: In part (a), we show the intersection graph corresponding to the segments shown in Fig. 3.5b. Each node represents a segment and there is an intersection edge (solid line) if the segments intersect. There is a via edge (broken line) if the corresponding segments have a via between them. The connected components formed by the intersection edges are coalesced into a single node, represented by an arbitrarily chosen node from that component as shown in part (b). This is done because coloring one node in a connected component forces a color on the remaining nodes. The via edges have a weight depending on the degree of the via, which we have taken as 6,2,1 for degree of 2, 3 and 4 respectively. The cluster graph shown in part (b) can be colored as shown to achieve maximum consensus.

immediately know that we have to color nodes *4* and *8* BLUE, because each of them has an intersecting edge with node *1*. We are, therefore, forced to assign a color to all the remaining nodes in the connected (by intersection edges) component of the graph if we assign a color to one of the nodes in that component. The connected components of the intersection graph shown in Fig. 3.6a are: {4,1,8,5}, {2,9,6},{3,7}, {10,26,21,23,14,15,22}, {24,17}, {11,16}, {19,12}, {20,25}, {18}. Since this is the case, it is convenient to coalesce all the nodes of a connected component into a single node, or, for convenience, we can simply represent all the nodes of a connected component by an arbitrarily chosen node of that component as shown in Fig. 3.6b. Assigning a color to the representative node forces a color on the remaining nodes. Note that for a two-layer assignment to be possible, the graph must not have odd cycles, because such graphs are obviously not two-colorable.

The via edges have weights which depend on the degree of the via upon which the corresponding segments are incident. They were chosen to be 6,2,1 for via degree 2,3 and 4 respectively. The weights were so chosen to keep them integral and representative of the contribution to the consensus when all the involved nodes assume the same color. For example, for a via of degree 4, there are four nodes involved with six edges between them. Therefore, each edge should have a weight of 1. When all four nodes have the same color, the via is removed and the contribution to the consensus is 6. Similarly, for a via of degree 3, three nodes are involved with three edges. Therefore, each edge should have a weight of 2 in order for the contribution to be 6, when all three nodes assume the same color. Again, for a via of degree 2, there are two nodes involved and one edge, which must, therefore, have a weight of 6. Note that a via edge $\{i, j\}$ will contribute a via, if the two nodes i and j assume different colors. Some of the edges, in Fig. 3.6, are marked with asterisks indicating that they

will contribute a via. For example, the edge {4,5} will contribute a via because nodes 4 and 5 cannot be colored the same. Similar is the case with the edges {21,22} and {22,23}. One of the edges {2,3} or {6,7} will contribute a via, because if we color 2 BLUE and 3 BLUE, 6 will be BLUE and 7 will be RED.

3.2.2 The Cluster Graph

Each node in the cluster graph is a representative of the nodes in the connected component of the intersection graph. The weight between the nodes of the cluster graphs represents the contribution to the consensus when the nodes are colored the same. The higher the consensus, the fewer the the number of vias. The weights are obtained from the intersection graph. Consider, for example, clusters 1 and 2. There are two via edges {1,2} with weight 6 and {8,9} with weight 2. If we color the representative nodes the same, the number of vias will be reduced by $1\frac{1}{3}$. Therefore, the total contribution to the consensus is 6+2 = 8, when the representatives are colored the same and -8 if they are colored differently. Consider clusters 2 and 3. If we color the representatives the same, edge {2,3} will not contribute a via but edge {6,7} will contribute a via because nodes 6 and 7 will be colored differently. Therefore, the total contribution to the consensus is 6-6 = 0, whether or not, we color the representatives with the same color. Similary, if we consider the clusters 2 and 10, there is one via edge {10,9} between them and this will contribute a part of a via, if we color the representatives with the same color. If we color the representatives differently part of a via will be reduced. Therefore, the weight is -2 and the contribution to the consensus is positive if the two nodes assume two different colors, represented by +1 and -1.

3.2.3 The Max Cut Formulation

So far, we have formulated the via-minimization problem in terms of maximizing the consensus function. We can directly work with this formulation and implement it on a neural network, which in our case, was a discrete Hopfield network. It is also possible to formulate the via minimization problem as the classical max cut problem. Given a graph $G = (V, E)$ with positive weights in the edges, the max cut problem is defined as the problem of finding a partition of $V = \{1, 2, \ldots, n\}$ into disjoint sets V_0 and V_1 such that the sum of the weights of the edges that have one endpoint in V_0 and the other endpoint in V_1 is maximal. The two vertex sets correspond to our two colors. We must first subtract a large enough constant from all our edge weights to make every weight negative or zero and then negate the resulting weight to get all positive weights as required by the max cut problem. The consensus maximization of the max cut problem can easily be derived by considering a 0-1 formulation of the max cut problem [34]. Let w_{ij} be the weights that have been calculated for the max cut problem as described earlier. Then the interconnection strengths W_{ij} can be shown to be $W_{ij} = -2w_{ij}$ and similarly the bias (external input) can be shown to be $I_i = \sum w_{ij}$, where, the sum is over the neurons adjacent to i. This process is carried out so as to give an *order-preserving* consensus function.

3.2.4 Experimental Results

We have experimented with both the max cut formulation and the direct consensus maximization formulation described earlier. Both the methods yield almost identical results for small problem instances (upto 25 nets). However, for larger problems, the maxcut formulation yields extremely poor results, although it appears that the two formulations ought to give identical results. The experimental results reported

here, were done using consensus maximization as described in Fig. 3.6. Compared to channel-width minimization, the neural network solution for the via minimization problem is more difficult, because the former was a constraint-satisfaction problem, whereas, in the latter problem there is an objective function to maximize, but, fortunately fewer neurons were involved. The optimality of the solutions cannot be ascertained easily. Table 3.2 shows the results of our experiments. As far as we know,

Problem Instance	Vias in Manhattan Model	Vias Before Minimization	Nodes in Cluster Graph	Minimum Vias
ex1	63	53	31	37
ex3b	131	152	70	94
ex3c	153	168	103	127
ex5	163	275	152	132
dif	302	635	255	235

Table 3.2: The above table shows our experimental results. There was a 15-40 percent reduction of vias. The number of nodes in the cluster graph is significant because it indicates the difficulty of finding an optimal solution. The number of *vias before minimization* column shows the number of vias obtained with a random coloring of the cluster nodes. Hopfield network was allowed to reach equilibrium 200 times, each time starting with a different initial state. The equilibrium state with the maximum consensus was noted, which yielded the smallest number of vias.

published data for the via minimization problem using the benchmark problem instances, do not exist. Xiong and Kuh [42], reported a solution with 335 vias for the Deutsch's Difficult problem. It was not mentioned whether or not it used doglegging, because, doglegging increases the number of vias. It should be remarked here that the deterministic heuristic they used, although not meant to be a neural network algorithm, has turned out to be a Hopfield network, iterated to equilibrium just once. In other words, they found a local minima. They did not perturb the initial state and retry as we have done. The problem was formulated as a max cut problem and the initial state was chosen by first sorting the edge-weights in descending order and

then selecting the edges in a greedy manner. In the light of these remarks, we feel that our results are definitely superior. In Fig. 3.7 and Fig. 3.8, we show the outputs from our via minimization program. For small-sized problems, the network seems to give optimal solutions. However, for larger problems with over 20 nets, it is difficult to ascertain the optimality of the solutions manually.

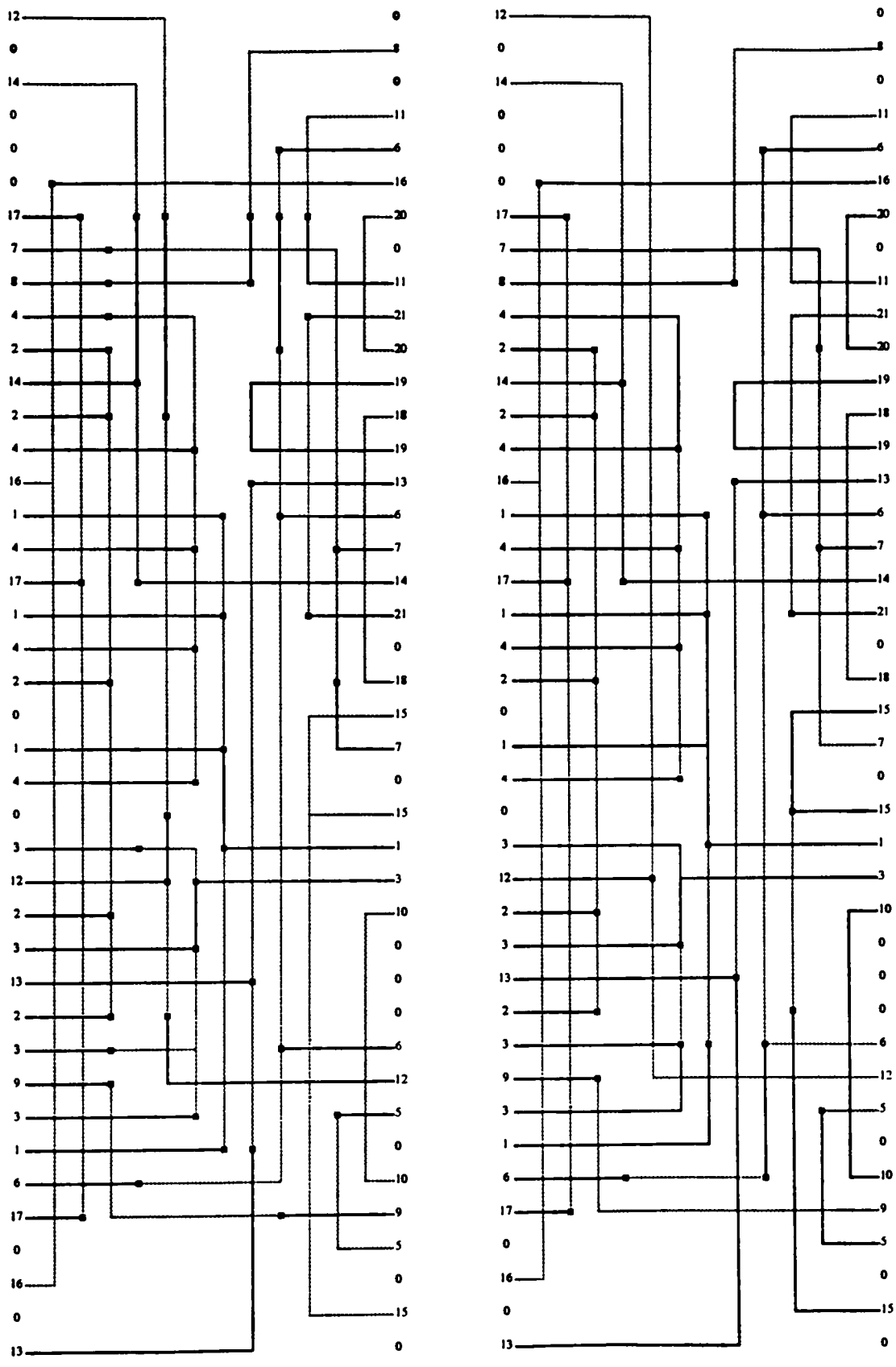


Figure 3.7: Via minimization of *ex1* with vias before minimization (left), and after minimization (right) obtained by our program. The initial track assignments are shown in Fig. 2.27. There were initially 63 vias in the Manhattan routing which were finally reduced to 37.

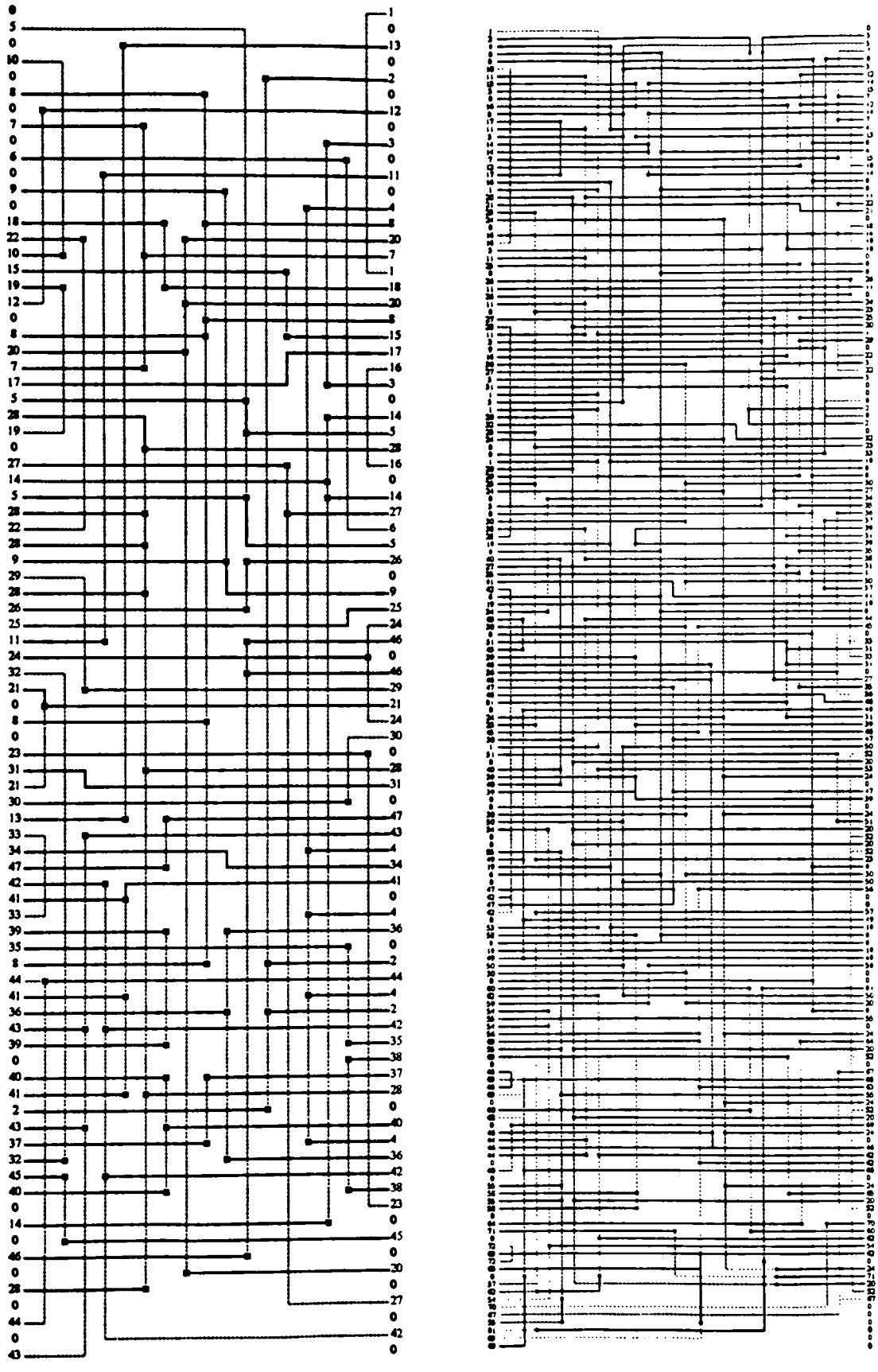


Figure 3.8: Minimized vias for *ex3b* (left), and *Deutsch's Difficult* Problem (right) obtained by our program. The track assignments are shown in Fig. 2.27. The number of vias were reduced from 131 to 94 for ex3b, and from 302 to 235 for *Deutsch's difficult*

Chapter 4

Conclusions

Because of past indifferent performances of neural networks in solving optimization problems, this work was initiated with some diffidence and abundant skepticism. The results previously obtained for the channel routing problem were not particularly encouraging. However, as the work progressed, the results obtained by our proposed neural networks became more and more impressive. It came as a complete surprise to us, when our proposed models were able to solve *Deutsch's Difficult* problem in two layers, because, a sequential heuristic developed by us previously could not produce an optimal solution even after considerable effort. Even if all our models are not regarded as "neural", they provide a framework for parallel processing. Our simulations can be run on parallel processors with 100% speedup because each processor can start its own simulation program with a different random number seed, complete the simulation and then send the result back to the host. There is no communication overhead, except when the job is done.

The first part of the thesis on channel-width minimization produced an efficient algorithm for assignment of tracks, generating cliques, etc., which can be of value in

any algorithm for channel-width minimization. It also proposed a channel decomposition method for multilayer routing. Further research needs to be done in this regard, since decomposition appears to be inherently important for multilayer routing. Our decomposition algorithm and variations of it should be compared to that proposed by Braun et. al [22] and used in Chameleon.

Our four-layer method can also be modified to produce a three-layer routing which is of great practical significance. This can be done by first producing a four-layer routing and then merging the vertical layers into a single layer, which will inevitably produce shorts, that need to be resolved by doglegging. Since our decomposition is non-deterministic, various decompositions could be tried and one that yields the best results could be retained.

Our method for determining *feasible* tracks can be used in other channel routing algorithms. Our estimate of the lower bound obtained by examining the interactions between the vertical and the horizontal constraint graphs can be very useful in branch-and-bound or the A* algorithms for reducing the size of the search space.

The second part of the thesis, which considered *constrained* via minimization, again considered the standard benchmark problems and produced via-minimized layouts. Since previous results are scanty, it is not clear, whether or not, we obtained optimal solutions. We can also regard our via-minimized solutions as "pseudo" two-dimensional channel routing. Two methods were proposed. Both methods were, however, not applied to the benchmark problems. It will be interesting to find out how they compare in practice. The cluster-graph based method is quite complicated, because, we first have to identify segments and introduce candidate vias. So, if the simpler method produces almost equally good results, one can use that in practice. Known heuristics should be implemented and compared to our results.

Although the thesis required the writing of 8,000 lines of code, it was worth the time and trouble. It certainly produced some useful results in channel routing and some insight into the paradigms of neural networks. In particular, it showed that for *constraint satisfaction* problems, optimization with the discrete Hopfield network may yield better results than the continuous Hopfield network. It also showed the importance of providing the neural network with all possible constraints.

Bibliography

[1] A. Hashimoto and S. Stevens, "Wire routing by optimizing channel assignment within large apertures," *Proc. of the Eighth Design Automation Conference*, 1971, pp. 455-476.

[2] M. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[3] T. Lengauer, *Combinatorial Algorithms for Integrated Circuits Layout*, Wiley, 1990.

[4] A. S. La Paugh, *Algorithm for Integrated Circuit Layout: An Analytic Approach*, Ph.D. Thesis, MIT, Tech. Rep. MIT/LCS/TR-248.

[5] T. G. Szymanski, "Dogleg channel routing is NP-complete," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-4(1), pp. 31-41, 1985.

[6] M. Sarrafzadeh, "Channel routing problem in the knock-knee model is NP-complete," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-6(4), pp. 503-506, 1987.

[7] T. Fujii, Y. Mima, T. Matsuda and T. Yoshimura, "A multilayer channel router with new style of over-the-cell routing," *Proc. of the 29th ACM/IEEE Design Automation Conference*, pp. 585-588, 1992.

[8] S. Natarajan, N. Sherwani, N. D. Holmes, and M. Sarrafzadeh, "Over-the-cell channel routing for high performance circuits," *Proc. of the 29th ACM/IEEE Design Automation Conference*, pp. 600-603, 1992.

[9] D. N. Deutsch, "A dogleg channel router," *Proc. of the Thirteenth Design Automation Conference*, pp. 425-433, June 1976.

[10] T. Yoshimura and T. S. Kuh, "Efficient algorithms for channel routing," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-1, pp. 25-35, 1982.

[11] R. L. Rivest and C. M. Fiduccia, "A greedy channel router," *Proc. of the 19th Design Automation Conference*, pp. 418-424, 1982.

[12] M. Burstein and R. Pelavin, "Hierarchical wire routing," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-2(4), pp. 223-234, 1983.

[13] B. W. Kernighan, D. G. Schweikert and G. Persky, "An optimum channel routing algorithm for polycell layouts of integrated circuits," *Proc. of the 10th Design Automation Workshop*, pp. 50-59, 1973.

[14] J. Wang and R. C. T. Lee, "An efficient channel routing algorithm to yield an optimal solution," *IEEE Trans. Comp.* vol. 39. no. 5, pp. 957-962, July 1990.

[15] Z. Lin, "An efficient optimum channel routing algorithm," *Proc. of the 28th ACM/IEEE Design Automation Conference*, pp. 1179-1183, 1991.

[16] V. Pitchumani and Q. Zhang, "A mixed HVH-VHV algorithm for three-layer channel routing," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-6(4), pp. 497-502, 1987.

[17] Y. K. Chen and M. L. Liu, "Three-layer channel routing," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-3(2), pp. 156-163, 1984.

[18] P. Bruell and P. Sun, "A greedy three-layer channel router," *Proc. of the International Conference on Computer-Aided Design*, pp. 298-300, 1985.

[19] W. Heyns, "The 1-2-3 routing algorithm or the single channel 2-step router on three interconnection layers," *Proc. of the 19th Design Automation Conference*, pp. 113-120, 1982.

[20] J Cong, D. F. Wong and C. L. Liu "A new approach to three- or four-layer routing," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-7(10), pp. 1094-1104, 1988.

[21] R. J. Enbody and H. C. Du, "Near-optimal n-layer channel routing," *Proc. of the 23rd IEEE Design Automation Conference*, pp. 709-714, 1986.

[22] D. Braun, J. L. Burns, F. Romeo, A. L. Sangiovanni-Vincentelli, K. Mayaram, S. Devadas, H. K. T. Ma, "Techniques for multilayer channel routing," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-7(6), pp. 698-712, 1988.

[23] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms and architectures," *Neural Networks* vol 1, no. 1, pp. 17-61, 1988.

[24] T. Kohonen, "An introduction to neural computing," *Neural Networks* vol 1, no. 1, pp. 3-16, 1988.

[25] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Nat. Acad. Sci. U.S.A*, vol. 79, pp. 2554-2558, 1982.

[26] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proc. Natl. Acad. Sci., USA*, vol. 81, pp. 3088-3092, 1984.

[27] Y. Takefuji, "A near optimum parallel planarization," *Science*, vol. 245, pp. 1221-1223, 1989.

[28] K. T. Sun and H. C. Fu, "A hybrid neural network model for solving optimization problems," *IEEE Trans. Comp.*, vol. 42, no. 2, pp. 218-227 Feb 1993.

[29] J. Hopfield and D. Tank, "Neural computation of decisions in optimization problems," *Biol. Cybernet.*, vol. 52, pp. 141-152, 1985.

[30] D. Tank and J. Hopfield, "Simple neural optimization networks: An A/D converter, signal decision circuit and linear programming circuit," *IEEE Trans. Circuits and Syst.*, vol. CAS-33, pp. 533-541, 1986.

[31] G. A. Tagliarini, J. F. Christ and E. W. Page, "Optimization using neural networks," *IEEE Trans, Comp.* vol. 40, no. 12, 1347-1358, 1991.

[32] I. Takefuji, *Neural Network Parallel Computing*, Kluwer Academic Publishers, 1992.

[33] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for Boltzmann machines," *Cognitive Science*, vol. 9, pp. 147-169, 1985.

[34] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines, A Stochastic Approach to Combinatorial Optimization and Neural Computing.* New York, Wiley, 1989.

[35] N. Funabiki and Y. Takefuji, "A parallel algorithm for channel routing problems," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-11, pp. 464-474, 1992.

[36] Pao-Hsu Shih and Wu-Shung Feng, "An application of neural networks on channel routing," *Parallel Computing*, vol. 17, pp. 229-240, 1991.

[37] R. Fujii, M. F. Tenorio and H. Zhu, "Use of neural nets in channel routing," *IJCNN*, pp. I-321-325, 1989.

[38] M. R. Zargham and M. R. Sayeh, "A neural network design for channel routing," *Proc. Optical Engg. Midwest '90*, pp. 202-208, 1990.

[39] K. T. Sun and H. C. Fu, "A neural network approach to restrictive channel routing problems," *Proc. Int. Conf. on Artificial Networks, ICANN*, 1992.

[40] N. J. Naclerio, S. Masuda and K. Nakajima, "The via minimization problem is NP-complete," *IEEE Trans. Comp.*, vol. 38, pp 1604-1606, 1989.

[41] K. Ahn and S. Sahni, "Constrained via minimization," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-12, pp. 273-282, 1993.

[42] X. Xion and E. S. Kuh, "The constrained via minimization problem for PCB and VLSI Design," *25th ACM/IEEE Design Automation Conference*, pp. 573-578, 1988.

[43] D. J. Haglin and S. M. Venkatesan, "Approximation results for the two-layer constrained via minimization problem," CAD, vol. 21, pp. 463-469, 1989.

[44] M. Sarrafzadeh and D. T. Lee, "A new approach to topological via minimization," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, CAD-8, pp. 890-900, 1989.

[45] M. Sarrafzadeh and D. T. Lee, "Topological via minimization revisited," *IEEE Trans. Comp.*, vol. 40, pp. 1307-1315, 1991.

[46] M. Stallman, T. Hughes and W. Liu, "Unconstrained via minimization for topological multilayer routing," *IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 970-976, 1990.

[47] S. S. Kim and C. M. Kyunn, "Via minimization using neural networks," *Journal of the Korean Institute of Telematics and Electronics*, vol. 27, pp. 129-136, 1990.

[48] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal*, 49(2):291-307, 1970.

Appendix A

Data Files

The two integers in the first line of each data set gives the number of columns and the number of nets. There then follows the net numbers for each column. All the nets on the top row are given first followed by the data for the bottom row.

```
13 9                                                    FENG5
1 5 7 0 7 9 9 8 1 2 3 3 2
5 6 5 4 8 8 9 6 3 0 4 2 0

12 10                                                   FENG6
0 1 4 5 1 6 7 0 4 9 10 10
2 3 5 3 5 2 6 8 9 8  7  9

12 10                                                   FENG7
1  2 3  0 2 4 5 0 3 6 7 7
9 10 1 10 1 9 4 8 6 8 5 6

12 10                                                   FENG8
1 2 0 3 4 0  5 4 1  6 7 0
8 8 7 3 2 9 10 9 0 10 6 5

13 10                                                   FENG9
0 1 2 3 1 4 0 4  0 2  5 6 6
7 8 3 8 3 9 9 7 10 5 10 4 5

15 13                                                   FENG10
1 2  3 4  0 3 5  0  6  1  7 8  8  9  9
5 7 13 0 13 4 2 11 12 12 11 6 10 11 10
```

Figure A.1: Some data sets from Shih and Feng [36] are shown above. Note that *Feng11* is the same as *ex1*.

```
41 21   ex1
0 13 0 17 9 23 33 0 17 34 33 32 31 32 20 9 10 21 34 0 31 22 10 0
22 1 3 16 0 0 0 9 19 7 0 16 14 7 0 22 0
19 0 21 0 0 0 24 10 13 4 2 21 2 4 23 1 4 24 1 4 2 0 1 4 0 3 19 2
3 20 2 3 14 3 1 9 24 0 23 0 20

90 45   ex3a
2 0 11 0 5 0 7 0 9 0 3 0 13 0
0 1 42 14 7 15 2 42 13 9 16 0 17 31 41 12 16 3 18 30 18 32 36 41
40 29 0 9 2 19 28 0 22 2 32 23 36 43 17 0 36 43 40 20 43 0 12 35
0 19 33 0 23 27 34 24 20 3 26 45 0 25 0 26 0 45 0 33 0 35 0 24 0
38 0 25
0 14 0 12 0 10 0 8 0 15 0 4 0 1
6 0 11 0 0 10 0 6 0 6 10 29 0 0 30 0 12 12 0 0 30 41 31 0 30 29 5
0 0 8 0 4 9 0 0 8 0 5 29 21 27 0 2 23 28 9 0 44 28 0 32 39 19 0
36 12 40 34 0 37 38 0 22 0 34 0 32 0 21 0 37 0 44 0 39 0

84 47   ex3b
1 0 13 0 2 0 12 0 3 0 11 0 4
8 20 7 1 18 20 8 15 17 16 3 0 14 5 28 16 0 14 27 6 5 26 0 9 25 24
46 0 46 29 21 24 30 0 28 31 0 47 43 4 34 41 0 4 36 0 2 44 4 2 42
35 38 37 28 0 40 4 36 42 38
23 0 45 0 20 0 27 0 42 0
0 5 0 10 0 8 0 7 0 6 0 9 0
18 22 10 15 19 12 0 8 20 7 17 5 28 19 0 27 14 5 28 22 28 9 29 28
26 25 11 24 32 21 0 8 0 23 31 21 30 13 33 34 47 42 41 33 39 35 8
44 41 36 43 39 0 40 41 2 43 37 32 45 40
0 14 0 46 0 28 0 44 0 43
```

Figure A.2: The benchmark data sets from Kuh and Yoshimura [10] are shown above. They collected the data from various sources. Note that in the data set *ex1*, the nets are not numbered sequentially. In our experiments, we renumbered them sequentially.

```
104 54     ex3c
1 0 13 0 2 0 12 0 3 0 11 0 7
3 4 15 14 3 2 1 19 6 0 0 19 17 20 2 25 9 20 0 28 21 22 23 25 11
22 0 24 10 15 27 0 24 23 1 28 5 0 0 31 30 28 36 39 52 23 33 35 0
34 52 1 30 8 0 0 15 43 8 41 0 34 43 15 32 48 25 44 25 0 49 15 0
50 0 47 53 50 49
31 0 24 0 34 0 47 0 51 0 29 0
0 4 0 10 0 5 0 9 0 6 0 8 0
15 4 6 17 16 14 2 19 9 18 16 11 18 26 0 15 10 5 21 0 26 27 23 28
0 8 5 7 29 0 12 30 23 37 0 13 0 35 36 33 37 38 0 40 0 28 38 0 43
30 41 44 52 0 42 32 52 30 48 18 42 39 0 51 18 0 54 53 0 45 40 0
46 0 45 50 47 54 0
0 45 0 46 0 48 0 50 0 54 0 35

119      57     ex4b
1 2 0 0 0 0 3 4 4 5 6 7 1 9 12 0 7 9 1 10 11 10 23 40 10 40 54 12
54 13 14 15 16 0 17 0 18 19 0 14 18 12 12 19 16 20 21 19 21 22 0
0 0 24 23 25 26 26 0 18 0 25 21 56 27 3 0 0 0 0 28 30 27 5 19 29
18 30 31 0 0 20 31 30 21 32 57 0 0 29 31 0 30 8 28 8 19 35 19 0 0
31 0 0 0 33 0 34 35 0 36 51 33 36 36 36 0 37 0
0 6 38 10 2 38 13 0 38 9 0 33 0 0 39 38 42 3 0 0 40 0 41 42 5 0 11
43 0 44 44 0 43 45 0 41 12 3 41 45 19 0 40 10 46 33 46 22 0 0 17 47
49 12 10 15 19 48 20 19 48 0 47 56 49 20 48 24 49 10 45 0 33 0 55
10 50 48 5 31 9 51 45 0 0 50 57 32 5 49 37 52 0 39 10 35 36 51 52
37 35 55 36 52 37 8 52 34 36 37 52 32 33 53 52 32 53 36 37
```

Figure A.3: Data Sets from Kuh and Yoshimura *ex3c, ex4b.*

```
130 63              ex5
1 0 3 0 5
1 2 44 3 0 0 0 0 4 0 2 5 0 2 0 0 2 0 0 2 0 6 0 0 0 0 0 0 2 7 0 0
0 0 3 0 0 8 0 0 9 0 10 11 12 13 0 0 0 1 0 0 14 0 63 0 0 16 3 14 15
16 17 18 0 19 20 21 0 0 14 22 14 0 23 0 24 25 19 26 17 27 28 29
30 19 31 59 26 31 32 0 33 0 34 29 35 21 28 21 36 27 35 36 37 24 56
62 22 34 36 37 38 39 0 0 0 0 0 0
62 0 24 0 31
0 2 0 40 0
0 0 40 41 42 43 44 45 4 9 46 0 1 0 47 48 0 46 49 41 49 47 43 47
13 42 49 13 50 51 6 11 7 52 45 48 0 49 53 15 54 50 18 51 0 55 52
32 54 8 16 53 17 8 63 19 47 10 0 17 26 0 0 0 5 14 20 0 5 31 56 0
14 57 0 35 33 0 0 0 0 0 0 0 0 55 0 0 0 0 57 36 0 12 0 23 0 0 58 0
0 34  59 30 25 60 0 22 38 56 58 61 0 34 39 24 58 58 60 61
0 35 0 14 0

175 72              dif
  0  3  5  7  9  5 12 14 15  7 12 14  7  4 13  8  6 15 18 14  8
  6 11 22 21  0 18 16 18 16  0  8  6 26 11  0 24 23 25 20  1 29
  0 22  3 22  3  0  0  9  2  9  2  0 32 23 33 19  6  8 30 27 34
 35 36 37 39 31 39 35 38 31  8 30 37 41 19  6 44 45  0 33 31 33
 31  0 27 35 36 48 49 31 39 46 47 50 52 20 53 24  0 47 39  0 24
 51 20 52 20 52 23  8 30 50 56  0  0 57 49 19  6  6 19 49 59  0
  0 61 50 30  8 55  0 24 64 20 52  0 67 68 63 55 24 52 20 69 24
  0 46 62 63 68  0 24 65 20 52  0 70 60 62 54 63  0 24 71 20 52
 67  0  0  0  0  0  0
  1  2  4  6  8 10 11 13  3  9 16  5 17 11  5 14 14  7 12 17 19
  1 20 21 23 24  0 16 10  3 11 25  0 26 11 26 11  0 27 28 11  3
  9 16 30 27  5 31  1  5  1 20 32 23 24  0  9  1 20 29 23 24  0
  3  8 30 38 28 19  6 40 27 35 41 42  6 19 34 43 30  8 31 43 39
 46 36 46 47 48 31  0 24 23 45 20  1 51  0 40 39 40 39  0  8 30
 50 54  0  0 55 49 19  6  0 47 42 47 42  0 53 58  6 19 49 50 30
  8 60 62 59 54 55 54 56 63 55 65  0 66 68 66 68  0 60 68  0 46
 44 46 44  0 69  0 55 58 55 58  0 64 71  0 72 63 72 63  0 57 62
 54 70 67 55 61 63 68
```

Figure A.4: Data Sets from Kuh and Yoshimura *ex5, Deutsch's Difficult Problem.*

```
12   ex1 solution
 6  10   7   7   2   4   2   5  10   1   3   8   5   9   3  12
11   1   5   1   3

17   ex3b solution
 1   6   3   4   7   2  12   9   8  16  14  17  13   3   5   1
 5  11  16  10  17  15   1   1   2   7   5  12  15   2  15  16
17   8   2   8   9   2  11  11  13  14  15  17  16   7  11

28    dif solution
21   9   8  20  19  16   2   4   3  28  22   5  18  17   1   6
24   1  20  23   5   2  26  11   5   1   7  28   5  14   6  10
 1  25   5   2   3  22  18  24  15  28  27  22  13  12  15   3
27  19   2   1  21  25  24  13  26  18  17   9   8  21  13   5
 6  28   2  27  28   3  15  28
```

Figure A.5: The track assignments for *ex1*, *ex3b* and *dif*,found by our channel-width minimization programs. These were used for via minimization and produced the output shown in Fig. [3.6] and Fig. [3.7]. The first number is the number of tracks used. The numbers following that are the track assignments for nets increasing sequentially. For example, the problem instance *ex1* required *12* tracks and nets *1*, *2*, *3*, ..., were assigned tracks *6*,*10*,*7*, ..., respectively.